Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Håkon Harnes

Deceptive bytes: An in-depth evaluation of WebAssembly obfuscation for evading crypto mining detection

Master's thesis in Computer Science
Supervisor: Donn Morrison
June 2023

NTNU
Norwegian University of
Science and Technology

Håkon Harnes

Deceptive bytes: An in-depth evaluation of WebAssembly obfuscation for evading crypto mining detection

Master's thesis in Computer Science
Supervisor: Donn Morrison
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Preface

The motivation for this thesis originates from the literature review conducted last semester as part of the *TDT4501 – Specialization Project* course. The literature review has since been formalised into a paper. A preliminary version is included in Appendix C for reference. Some of the background sections of this thesis are derived from the literature review.

Acknowledgements

I would like to acknowledge those who have helped me while writing this thesis. Their guidance, support, and encouragement have been invaluable.

First and foremost, my deepest gratitude goes to my supervisor, Associate Professor Donn Morrison. From the very beginning, he has believed in my capabilities, granting me the liberty to freely explore my ideas and interests. His guidance and support have been invaluable, and I am grateful for the opportunity to work with him. Thank you, Donn, for placing your trust in me and for your patience and encouragement throughout the process.

I also extend my gratitude to all the researchers and academics who generously responded to my numerous emails and inquiries. My sincere thanks goes to Christian Collberg, the author of Tigress, for the countless email exchanges helping me debug Tigress. Likewise, I am thankful to Javier Cabrera Arteaga, a PhD candidate at KTH, for his valuable insights and for helping me re-train MINOS. Further, I am grateful to Alano Romano for his assistance with MinerRay, Daniel Lehmann for his insights on the V8 engine, and Robert Aboukhalil for introducing me to Biowasm, which helped extend the dataset.

Moreover, I would like to thank my friends who have inspired and supported me. A special thanks goes to Vetle Harnes for the many hours spent at school working together, discussing ideas, and, of course, for consistently outplaying me in our many chess matches. I would also like to sincerely thank Tor Henrik for his dedication in proofreading this thesis.

I am deeply grateful to my significant other, Kristine Saxe Grødal, for her valuable feedback and encouragement throughout the process. Thank you for your patience, for always being there for me, and, of course, for those perfectly timed and much-needed cups of coffee.

Lastly, I would like to thank my friends and family for their love and support throughout my studies. Their continuous encouragement and belief in my abilities have been a tremendous source of motivation.

Abstract

WebAssembly is a low-level bytecode language that allows high-level languages like C, C++, and Rust to be executed in the browser at near-native performance. It has already seen extensive adoption and is now natively supported in all modern browsers. However, its adoption also presents new challenges in security. One of the most notable is drive-by mining attacks, where websites use the computing resources of their visitors to mine cryptocurrency without their knowledge or consent. A plethora of approaches have been proposed to detect drive-by mining, yet there is a lack of research on how these detection methods can be sidestepped.

This thesis provides an in-depth evaluation of code obfuscation for WebAssembly, a largely unexplored topic. The objective is to evaluate how well code obfuscation can disguise the underlying nature of the code and evade drive-by mining detection. We perform the most comprehensive evaluation of WebAssembly obfuscation to date, applying obfuscation at multiple abstraction levels – including source code, LLVM bitcode, and WebAssembly binaries. In this evaluation, we leverage existing obfuscation methods as well as develop novel ones. We not only evaluate the impact of obfuscation on the WebAssembly binaries but also how the resulting native code produced after compilation in the browser is affected. Moreover, we, for the first time in the literature, investigate to what extent obfuscation applied to WebAssembly can be reversed through automatic de-obfuscation and how this affects the detection capabilities. Lastly, we evaluate the overhead introduced by obfuscation, measuring the space and time overhead.

The results suggest that obfuscation can successfully evade state-of-the-art drive-by mining detectors. However, the effectiveness largely depends on the specific obfuscation transformation, detection method, and crypto mining algorithm. Moreover, we find that obfuscation can be partially reversed through automatic de-obfuscation, but the degree of success varies depending on the specific transformation. The native code generated from the obfuscated WebAssembly binaries tends to increase in size and, as a result, decrease in performance. Despite the significant performance overhead for some transformations, we show how obfuscation can be used for evading detection with minimal overhead in real-world scenarios.

These results offer insights into how researchers can develop obfuscation-resilient detection methods, thereby addressing the limitations of current methods. Moreover, we provide novel obfuscation and de-obfuscation methods, as well as an extensive dataset of nearly 50 000 obfuscated WebAssembly binaries for researchers to extend and explore.

Sammendrag

WebAssembly er et lavnivå bytekode-språk som lar høyere nivå språk som C, C++ og Rust kjøre i nettleseren med høy ytelse. Det brukes allerede i stor grad og støttes i alle moderne nettlesere. Men WebAssembly introduserer også nye utfordringer innen sikkerhet. En av de mest bemerkelsesverdige er såkalte "drive-by mining" angrep, hvor nettsider utnytter brukernes maskinvare for å utvinne kryptovaluta uten deres kjennskap eller samtykke. Det er utviklet en rekke metoder for å oppdage drive-by mining, men det er foreløpig lite forskning på hvordan disse metodene kan omgås.

Denne oppgaven utfører en grundig evaluering av kodeobfuskasjon for WebAssembly, et lite utforsket område i forskningslitteraturen. Målet er å undersøke til hvilken grad kodeobfuskasjon kan skjule den underliggende hensikten til koden og dermed omgå drive-by mining deteksjon. Vi gjennomfører den mest omfattende studien av kodeobfuskasjon for WebAssembly til nå, og anvender obfuskasjon på flere abstraksjonsnivåer – inkludert C kildekode, LLVM bitkode, og WebAssembly binærfiler. I denne prosessen tar vi i bruk en rekke eksisterende verktøy og utvikler også nye. Vi evaluerer effekten av obfuskasjon på WebAssembly binærfiler og hvordan den resulterende maskinkoden påvirkes etter kompilering i nettleseren. Videre undersøker vi, for første gang i litteraturen, i hvilken grad obfuskasjon som anvendes på WebAssembly kan reverseres gjennom automatisk de-obfuskasjon og hvordan deteksjonsevnen påvirkes av dette. Avslutningsvis ser vi på i hvilken grad obfuskasjon reduserer ytelsen, både når det gjelder plass- og tidskompleksitet.

Resultatene viser at obfuskasjon kan omgå state-of-the-art drive-by mining detektorer. Imidlertid er effektiviteten i stor grad avhengig av den spesifikke obfuskasjonstransformasjonen, deteksjonsmetoden og kryptovaluta-utvinningsalgoritmen. Videre finner vi at obfuskasjon kan delvis reverseres gjennom automatisk de-obfuskasjon, men graden av suksess er avhengig av den spesifikke obfuskasjonstransformasjonen. Maskinkoden som produseres av de obfuskerte WebAssembly binærfilene har en tendens til å øke i størrelse og oppnår også dårligere ytelse. Selv om reduksjonen i ytelse er betydelig for enkelte transformasjoner, viser vi hvordan obfuskasjon kan brukes i praksis for å omgå drive-by mining deteksjon med minimal ytelsesreduksjon.

Disse resultatene gir innsikt i hvordan forskere kan utvikle deteksjonsmetoder som er resistente mot obfuskasjon, og dermed adressere begrensningene ved eksisterende metoder. I tillegg har vi utviklet nye metoder for obfuskasjon og de-obfuskasjon, samt et omfattende datasett med nesten 50 000 obfuskerte WebAssembly binærfiler som kan anvendes i videre forskning.

Contents

List of Figures

List of Tables

Listings

Acronyms

ARPANET Advanced Research Projects Agency Network

ASIC Application-specific integrated circuit

AST Abstract syntax tree

CFG Control flow graph

CIL C intermediate language

CNN Convolutional neural network

COM Component object model

DFG Data flow graph

DOM Document object model

DTW Dynamic time warping

FN False negative

FP False positive

IoT Internet of Things

ISA Instruction set architecture

JIT Just in time

MBA Mixed boolean-arithmetic

NaCl Native Client

NTNU Norwegian University of Science and Technology

OLLVM Obfuscator-LLVM

PE Portable executable

pNaCl Portable Native Client

RF Random forest

SOP Same-origin policy

SVM Support vector machine

TN True negative

TP True positive

VM Virtual machine

WABT WebAssembly binary toolkit

WASI WebAssembly system interface

Chapter 1

Introduction

1.1 Motivation and background

The Internet is continually growing more powerful and complex. Today, it hosts a variety of sophisticated web applications ranging from social media platforms to games, streaming services, and photo editors. The advantage of these web applications over native software lies in their availability; they require no installation, are always up-to-date, and can be conveniently accessed and shared via hyperlinks. Moreover, they offer portability, operating seamlessly across different devices, operating systems, and hardware architectures. This development marks a significant departure from the Internet's initial use as a tool primarily for academics and researchers to share information and collaborate on projects.

The technological capabilities enabling the performance feats of web applications primarily hinge on browsers' ability to not only *display* website content but to also *execute* application code, providing interactive web experiences. JavaScript has been pivotal in this regard, enabling the development of dynamic web applications. Although JavaScript was developed in a mere ten days, its adoption has been extensive, spanning not only client-side web development but also server-side via Node.js [127] and desktop applications through the Electron framework [61]. Indeed, JavaScript is one of the most used programming languages, with 98% of all websites using it [163].

JavaScript, initially positioned as a minor counterpart to languages like Java, has exceeded initial expectations and, arguably, what its initial design intended. The ever-increasing demands of web applications have revealed the inherent limitations of JavaScript, predominantly due to its dynamic language features. Its performance is largely dependent on complex just in time (JIT) compilers, meaning maintaining consistent performance proves complex [153] and raises potential security concerns [79]. While the human-readable text format is convenient for developers, it proves to be less efficient as a code distribution format. With the mean website today containing nearly 500kB of JavaScript code [75], the time invested in parsing and compiling this code significantly increases

the load times of websites [36]. An increasing trend sees JavaScript code being *generated* from other languages rather than being *written* directly.[1] As Atwood's Law states, *"Any application that can be written in JavaScript, will eventually be written in JavaScript"* [50]. Inadvertently, JavaScript has become the primary code format for the Web — not by design, but by circumstance.

Over the years, several technologies have been developed to address the inherent limitations of JavaScript. This includes early attempts like Java [46], Flash [45], and ActiveX [44], to more recent attempts such as Google's Native Client (NaCl) [77] and Microsoft's asm.js [117]. Despite their novelty, they fell short of achieving extensive adoption due to security, compatibility, and performance issues. Consequently, their usage has dwindled over the years, further solidifying JavaScript's position as the predominant language of the Web.

In 2015, a consortium of leading technology companies – including Mozilla, Microsoft, Google, and Apple – announced they were working on a novel low-level bytecode language for the Web. WebAssembly, a response to the limitations that have plagued preceding technologies, is designed to be efficient, fast, and secure [80]. It allows high-level languages, like C, C++, and Rust, to be executed in the browser at near-native performance. Already, it has seen extensive adoption, with all major browsers supporting it in 2017, and as of May 2023, it is supported by 96% of all browser installations [168]. Notably, it earned formal approval as a web standard in 2019 [173]. As such, for the first time in over two *decades*, a new language has been introduced to the Web, standing shoulder-to-shoulder with JavaScript as a natively supported language.

WebAssembly was initially designed to complement JavaScript, specifically for the computationally intensive components of web applications. To that end, it has already been successful. Established corporations like Figma, eBay, and Google have adopted WebAssembly to improve performance [67, 60, 59]. WebAssembly serving as a compilation target for other languages has also been a significant factor in its adoption, allowing applications that were once desktop-only, such as Google Earth, Unity, and Photoshop, to be ported to the Web [59, 167, 22]. Beyond that, it has also been extended outside the Web; to desktop applications [112], mobile devices [137], cloud computing [66], blockchain virtual machines (VMs) [181, 63, 123], Internet of Things (IoT) [101, 109], embedded devices [151], and even stand-alone runtimes [178, 179].

Nonetheless, WebAssembly has also been exploited for malicious purposes. The most prominent example of this is drive-by mining, an attack strategy that exploits a website visitor's hardware resources to mine cryptocurrencies without their knowledge or consent. Such an approach has become feasible with the advent of cryptocurrencies like Monero [114] and VerusCoin [169], which can be mined using standard consumer-grade CPUs [113]. Although drive-by mining 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 drive-by mining incidents in 2018 [2]. By 2019, more than half of all websites containing WebAssembly used it for drive-by mining [120]. Reports from the first three quarters of 2022 confirm the steady growth of drive-by mining [94], indicating that it remains a pervasive problem.

[1] Such as TypeScript, or even asm.js for compiling C/C++ to JavaScript.

In response to the escalating threat of drive-by mining, a plethora of detection methods have been proposed. Some methods rely entirely on static analysis of the WebAssembly code [122, 146, 95, 145], leveraging techniques such as signature matching, control flow graph (CFG) analysis, and neural networks. Conversely, other methods use dynamic analysis [176, 143, 93, 92, 18], concentrating on behavioural characteristics like processor and memory usage, network traffic, and JavaScript events. Evidently, the literature is not lacking in approaches for identifying drive-by mining attacks.

Surprisingly, there is limited research on how the aforementioned detection methods can be sidestepped. Obfuscation, the process of making a program harder to analyse through applying various code transformations, has proven to be an effective evasion strategy for malware detection [115, 133, 107]. Although obfuscated WebAssembly binaries are a common occurrence on the Web, the subject of WebAssembly obfuscation has been scarcely explored. A short paper from 2021 merely touched upon this [17], and while conducting my research, two impending papers surfaced with a similarly limited focus [25, 103], only considering a select few obfuscation and detection methods. This recent burst of interest underscores the timeliness and relevance of the issue and the apparent gap in the literature.

1.2 Objectives and research questions

The primary objective of this thesis is to investigate and understand the application, effectiveness, and implications of code obfuscation for WebAssembly. Specifically, we evaluate how well obfuscation can disguise the underlying nature of the code and evade drive-by mining detection. Moreover, we delve into the extent of reversibility of obfuscation through automatic de-obfuscation and the resulting impact on detection accuracy. We also quantify the overhead introduced by obfuscation, both in terms of code size and hash rate, and evaluate whether the overhead is justifiable given the obfuscation advantages.

Guided by these objectives, the following research questions are investigated in this thesis:

- **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 drive-by mining detectors, and which transformations are the most effective?
- **RQ3 – Reversibility** How resilient are the transformations to automatic de-obfuscation, and how does de-obfuscation affect the detection accuracy?
- **RQ4 – Overhead** To what extent do the transformations contribute to overhead in terms of code size and hash rate?

1.3 Contributions

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

- We perform an exhaustive evaluation of code obfuscation for WebAssembly across a diverse dataset, including both malicious and benign applications. Our evaluation includes multiple obfuscation methods applied at several abstraction levels, a first in the literature. The insights derived from this assessment serve as a valuable resource to further advance detection methods.
- We investigate how obfuscation can be used to disguise the underlying nature of the code and evade state-of-the-art drive-by mining detectors.
- We build and release a novel obfuscation method for WebAssembly, emcc-obf,[2] by leveraging existing obfuscation tools. Moreover, we compare this method to existing obfuscation methods employed in the literature.
- We conduct an empirical study on WebAssembly obfuscation reversibility and resilience to automatic de-obfuscation, a perspective not previously explored in the literature.
- We quantify the overhead introduced by obfuscation, covering factors like code size and hash rate, differentiating between the CryptoNight variants. This granular exploration is a novelty in the existing body of literature.
- We compile a comprehensive dataset of nearly 50 000 obfuscated WebAssembly binaries spanning a diverse range of use cases.[3] This includes all the prominent CryptoNight variants, making it a significant asset for subsequent studies.

1.4 Scope and limitations

This research focuses primarily on obfuscation for WebAssembly, specifically in the context of drive-by mining. We investigate a variety of obfuscation methods and assess their effectiveness, reversibility, and the overhead they introduce.

However, there are several boundaries and limitations to the scope of this thesis:

- The focus of this thesis is on the obfuscation of WebAssembly with an emphasis on drive-by mining. While some of the findings could potentially be relevant to reverse engineering, we do not investigate this extensively. The highly subjective nature of reverse engineering studies, which often require many participants, is beyond the scope of this thesis.

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

- The evaluation is limited to individual obfuscation transformations. While combinations of transformations could potentially provide additional insights, the vast number of possible permutations would significantly increase the size of the dataset, rendering the study infeasible in practice.
- The performance overhead assessment is restricted to the crypto mining binaries within our dataset. Implementing a common benchmark to evaluate the overhead across all application categories is beyond the scope and focus of this thesis.
- Our research employs current, state-of-the-art static detection methods. As the field is rapidly evolving, new techniques may emerge that could alter the findings of this thesis. Additionally, we do not consider dynamic detection methods, which are not widely adopted due to the runtime overhead they introduce.
- To our knowledge, there are no WebAssembly de-obfuscation tools. As such, we resort to compiler optimisations as a substitute to remove the obfuscation. This approach may not perfectly capture the effectiveness of true de-obfuscation techniques that might be developed in the future.

These limitations should be taken into account when interpreting the findings of our thesis. They also highlight areas for potential future research in the field of WebAssembly obfuscation.

1.5 Thesis structure

This thesis is structured as follows:

- **Chapter 2 – Background** This chapter provides an overview of theoretical concepts relevant to this thesis. It traces the historical development leading to the creation of WebAssembly, offers an introduction to WebAssembly and its ecosystem, and delves into the issue of drive-by mining. It also presents a variety of detection methods and describes the concept of obfuscation, concluding with an exploration of code similarity.
- **Chapter 3 – Methodology** This chapter describes the experimental framework and implementation specifics of the thesis, including aspects such as system configuration, the dataset, and the methods used for obfuscation and detection. Lastly, it presents the metrics used to assess the research questions.
- **Chapter 4 – Results** This chapter presents the results of the experiments. The results are presented in line with the research questions, focusing on the effectiveness, detectability, reversibility, and overhead of the obfuscation methods.
- **Chapter 5 – Discussion** This chapter reflects on the main findings, aligning them with existing literature and offering potential explanations and implications. Moreover, we highlight the limitations of the thesis.

- **Chapter 6 – Conclusion** The final chapter provides a summary of the findings, discusses the implications of the research, suggest areas for future research, and concludes the thesis.

The appendices are structured as follows:

- **Appendix A – wasm-obf** This appendix contains the source code of the experiments conducted in this thesis and the associated experimental data.
- **Appendix B – emcc-obf** This appendix contains the source code and build scripts for the novel obfuscation tool developed in this thesis, emcc-obf.
- **Appendix C – Analysis Techniques for WebAssembly** This appendix contains the preliminary version of the paper derived from the literature review conducted last semester.

Chapter 2

Background

This chapter presents the theoretical knowledge required to understand the thesis.[1] Starting with Section 2.1, it offers an overview of the Internet's history and JavaScript's significance in web applications, as well as its limitations. Then, Section 2.2 presents prior attempts at implementing low-level code on the Web and their inherent limitations. Section 2.3 introduces WebAssembly, its security mechanisms, and its surrounding ecosystem. The concept of drive-by mining is explained in Section 2.4, followed by a presentation of the corresponding detection methods in Section 2.5. Section 2.6 details the theory and tools of obfuscation, alongside a review of existing attempts in obfuscating WebAssembly. Lastly, Section 2.7 presents the concept of code similarity, complexity metrics, and dynamic time warping.

2.1 The Internet

The origins of the Internet trace back to the 1960s, with the establishment of the Advanced Research Projects Agency Network (ARPANET), a project funded by the U.S. Department of Defense [19]. This ambitious initiative aimed to facilitate communication between universities and research institutions, utilising a then-novel concept called packet-switching [51]. Few would have thought back then that this project would eventually evolve into the Internet as we know it today. It was not until the 1990s, with the advent of the World Wide Web and Internet browsers such as Netscape and Internet Explorer, that the Internet became widely accessible to the general public [49]. Suddenly, the Internet was no longer confined to academia and research, and static, text-based web pages no longer sufficed. The Internet was in dire need of a catalyst, something that could breathe life and dynamic responsiveness into it.

That catalyst emerged in the form of JavaScript, a programming language enabling developers to create interactive web pages. In the spring of 1995, while at Netscape, Brendan Eich was given the

[1] Some sections are derived from the literature review conducted last semester.

formidable task of creating a language that would complement Java, enhance user interactivity, and change the face of web development [43]. In just ten days, Eich managed to design JavaScript – a feat that forever altered the course of the Internet. Following its release, the Internet transformed from a novelty into a necessity, enabling information sharing, connectivity, and entertainment, among numerous other aspects of everyday life. Two decades after those ten days in 1995, 98% of all websites on the Web use JavaScript [163], and it has even been extended to the server-side via Node.js [127] and desktop applications through the Electron framework [61].

Despite JavaScript's widespread adoption, it has some inherent limitations that have become apparent as web applications become more resource-demanding. Being a high-level language, JavaScript abstracts away details of the underlying hardware, and its dynamic typing means that the type of variables is determined at runtime. Therefore, optimal performance is dependent on intricate just in time (JIT) compilers, such as SpiderMonkey in Firefox or V8 in Google Chrome [153]. However, the complexity of these JIT compilers also presents potential security vulnerabilities [79]. Since JavaScript is an interpreted language, it must be parsed and interpreted every time it is executed. Given that the median website today contains nearly 500kB of JavaScript code [75], parsing and compiling this code contributes significantly to website loading times [36].

Moreover, an increasing trend sees JavaScript code being *generated* from other languages rather than *written* directly. Although an impressive feat of engineering, this can only be seen as a workaround to the limitations of JavaScript rather than a solution. Undoubtedly, JavaScript has transcended its original design and, arguably, its intended purpose. It has been positioned as the primary coding format for the Web, but not because it was designed for it.

2.2 Attempts at low-level code on the Web

Shortly after JavaScript's release, Microsoft developed the ActiveX framework in 1996 [44]. It allowed developers to embed signed x86 binaries through ActiveX controls. These controls were built using the component object model (COM) specification, which was intended to make the controls platform-independent. However, ActiveX controls contained compiled x86 machine code and calls to the standard Win32 API, restricting them to x86-based Windows machines [44]. Additionally, they were not run in a sandboxed environment, consequently allowing them to access and modify system resources. In terms of security, ActiveX relied entirely on code signing and thus did not achieve safety through technical construction, but rather through a trust model.[2]

Aimed to address the security issues of preceding approaches, Native Client (NaCl) was introduced by Google in 2011 [77]. It allows native code to be executed on the Web in a sandboxed environment. This sandboxing model enables the coexistence of NaCl code with sensitive data within the same process. However, NaCl is specifically designed for the x86 architecture, limiting its portability. To address this limitation, Google introduced Portable Native Client (pNaCl) in 2013 [33]. pNaCl builds upon NaCl's sandboxing techniques but uses a stable subset of LLVM bitcode as an inter-

[2]Developers had to register with Verisign and sign a contract *promising* not to develop malware.

changeable format, enabling the portability of applications across different architectures. However, pNaCl does not significantly improve compactness and still exposes details specific to compilers and architectures, such as the layout of the call stack. NaCl and pNaCl are exclusively available in Google Chrome, thus limiting the portability of the applications that use them.

Asm.js, which was introduced by Mozilla in 2013 [117], is a strict subset of JavaScript that can be used as an efficient compilation target for high-level languages like C and C++. Through the Emscripten toolchain [62], these languages can be compiled to asm.js and subsequently executed in modern JavaScript execution engines. As such, it benefits from sophisticated JIT compilers, enabling near-native performance. However, the strict subset nature of asm.js means that extending the language with new features, such as `int64`, requires first extending JavaScript and then ensuring compatibility with the asm.js subset. Even then, it can be challenging to implement these features efficiently.

It is also worth noting that Java and Flash were among the first technologies to be used on the Web, being released in 1995 and 1996, respectively [46, 45]. They offered managed runtime plugins; however, neither was capable of supporting high-performance, low-level code. Moreover, their usage has declined due to security vulnerabilities and performance issues.

2.3 WebAssembly

In 2015, a consortium of leading technology companies – including Mozilla, Microsoft, Google, and Apple – announced they were working on WebAssembly,[3] a low-level bytecode language for the Web [14]. It was designed to address the compatibility and security issues that have plagued preceding technologies while providing near-native performance [80]. By 2017, it was natively supported by all major browsers [80], and in 2019, it was formally standardised [173]. As of May 2023, WebAssembly showcases an impressive 96% support across all browser installations [168]. Today, WebAssembly stands alongside JavaScript as a natively supported language on the Web.

Initially, WebAssembly was intended to complement JavaScript, specifically for the computationally intensive components of web applications. In that, it has already been successful. Established corporations like Figma, eBay, and Google have adopted WebAssembly to improve performance [67, 60, 59]. Applications that were once desktop-only, such as Google Earth, Unity, and Photoshop, have been ported to the Web using WebAssembly [59, 167, 22]. Even AutoCAD, with its three-decade-long codebase, has been made available in the browser [30].

Despite its name, WebAssembly is not just limited to the Web. Its usage is expanding to desktop applications [112], mobile devices [137], cloud computing [66], blockchain Virtual Machines (VMs) [181, 63, 123], Internet of Things (IoT) [101, 109], embedded devices [151], and even standalone runtimes [178, 179].

[3] Sometimes abbreviated to Wasm

Without a doubt, WebAssembly, although still evolving, has already gained considerable momentum. It is set to become a crucial platform for computing, not only for the Web but for a broad range of applications in the years to come. Solomon Hykes, the founder of Docker, encapsulated its importance by stating: *"If WASM+WASI existed in 2008, we would not have needed to create Docker. That's how important it is. Webassembly on the server is the future of computing"* [88].

2.3.1 Language concepts

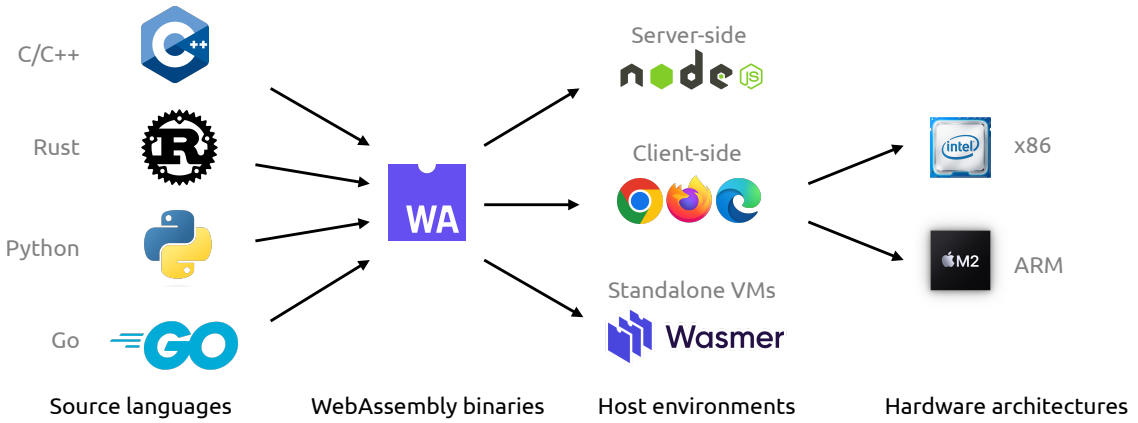


Figure 2.1: WebAssembly serves as the intermediate bytecode bridging the gap between multiple source languages and host environments. The host environments compile the WebAssembly binaries into native code for the specific hardware architecture.

Overview WebAssembly is commonly referred to as a compilation target, a format into which source code written in various languages can be compiled. As illustrated in Figure 2.1, WebAssembly acts as an intermediate step that enables interoperability between source languages and hardware architectures. As a bytecode, it provides a universal language into which different types of source code can be compiled, which can then be executed in different host environments. For instance, 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 it into native code for the specific hardware architecture.

Modules WebAssembly programs are organised into modules which serve as the fundamental units of deployment, loading, and compilation. These modules contain definitions for types, functions, tables, memories, and globals, which can be defined within the module or imported from the host environment. Similarly, they can be exported from the module under potentially multiple names and then imported by the host environment. Functions defined within the module have a corresponding function body located in the code section, whereas locally defined globals have an initialisation expression. Figure 2.1 contains an overview of the sections commonly found within a WebAssembly module.

ID	Section	Description
0	Custom	Debugging information or third-party extensions
1	Type	Function signatures of the functions defined within the module
2	Import	Functions, tables, memory, and global variables imported by the module
3	Function	Functions defined in the module. The local variables and the body of the functions are encoded in the code section
4	Table	Tables defined in the module. Contains the function table, which is used for indirect function calls
5	Memory	Linear memories used within the module
6	Global	Global variables used within the module
7	Export	Functions, tables, memory, and global variables that can be accessed by JavaScript
8	Start	The function index of the start function that is automatically invoked when the module is instantiated after tables and memories have been initialised
9	Element	List of elements used for initialising tables
10	Code	Local variables and the body of functions
11	Data	Data used for initialising memory

Table 2.1: Sections of a WebAssembly module.

Text and binary format Modules can be represented in two formats; the binary format (`wasm`) and the human-readable text format (`wat`). The binary format, being a compact representation, is designed for efficient network transmission and parsing. It is usually generated by compilers and instantiated in runtimes. The corresponding human-readable text format is designed for debugging, testing, and occasional manual editing, similar to native assembly languages. It is possible to convert between these two formats using tools such as `wasm2wat` and `wat2wasm` from the WebAssembly binary toolkit (WABT). Figure 2.2 shows the source code of a simple program, the human-readable `wat` format, and the corresponding `wasm` format.

Types Functions, instructions, and variables are statically typed, meaning their types are determined at compile time. There exist only four primitive values types: 32-bit and 64-bit integers (`i32`/`i64`) and single-precision and double-precision floating-point numbers (`f32`/`f64`). Aggregate types, such as classes, objects, and arrays, are not natively supported. Instead, during compilation, such types are transformed into primitive types and stored in memory.

Stack and variables Modules are executed on a stack-based virtual machine (VM). More specifically, instructions pop their inputs from and push their results to an implicit evaluation stack. There are no registers in this system, but individual values can be stored in global variables that are visible to the entire module or in local variables that are only visible to the current function. The evaluation stack, global variables, and local variables are all managed by the VM.

```
int square(int x)
{
  return x*x;
}
```

Listing 2.1: Source code.

```
(func (param i32) (result i32)
    local.get 0
    local.get 0
    i32.mul
)
```

Listing 2.2: `wat` format.

```
20 00 | local.get 0
20 00 | local.get 0
6c    | i32.mul
0b    | end
```

Listing 2.3: `wasm` format.

Figure 2.2: Example of a function compiled to WebAssembly. The source code, the equivalent human-readable format (`wat`), and the binary format (`wasm`) are shown.

2.3.2 Security

Environment Modules execute within a sandboxed environment separated from the host environment using fault isolation techniques. This implies that modules execute independently and cannot escape the sandbox without going through the appropriate APIs. For instance, WebAssembly modules in a web browser have no direct access to the document object model (DOM), but must use JavaScript APIs to interact with it. Additionally, each module is subject to the security policies of its embedding, such as the same-origin policy (SOP) enforced by web browsers [172], which restricts the flow of information between web pages of different origins. In the case of standalone WebAssembly runtimes with operating system support, the module must use the appropriate APIs to access system resources such as files, for instance, through the WebAssembly system interface (WASI).

Memory Unlike native binaries, which have access to all of the memory allocated to the process, WebAssembly modules are restricted to a contiguous region of memory known as linear memory. This memory is untyped and byte-addressable, and its size is determined by the data present in the binary. The size of linear memory is always a multiple of a WebAssembly page, which is 64KiB. When a WebAssembly module is instantiated, it uses the appropriate API call to create the necessary memory objects needed for its execution. The JavaScript engine or system runtime then creates a managed buffer, like an `ArrayBuffer`, to store the linear memory. This means that the WebAssembly module accesses the physical memory indirectly through the managed buffer, which ensures that it can only read and write data within a limited area of the memory.

Control flow integrity WebAssembly features a structured control flow, unlike other assembly-like languages [56]. Instructions in a function are organised into well-nested blocks. Branches can only jump to the end of surrounding blocks and only inside the current function. Multi-way branches can only target blocks that are statically designated in a branch table. Unrestricted gotos to arbitrary addresses are not possible. Data in memory cannot be executed as bytecode instructions, preventing attacks like shellcode injection or abuse of indirect jumps. The execution semantics guarantee the safety of direct function calls through the use of explicit function section indexes and returns through a protected call stack. The type signature of indirect function calls is checked at runtime. Additionally, the LLVM compiler infrastructure includes a built-in implementation of fine-grained control flow integrity, which has been extended to support the WebAssembly target [180].

2.3.3 Ecosystem

Compilation and 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 [35, 70]. Further enhancing this, Emscripten, a toolchain built on Clang and LLVM, is capable of porting C and C++ code to the Web using WebAssembly [62]. In addition to compiling C and C++ code to WebAssembly, it also produces the corresponding JavaScript "glue" code. The JavaScript code is responsible for instantiating the WebAssembly module, as well as for providing the necessary functionality for it to interact with the host environment. For instance, 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 [147]. Even garbage collected languages like C# [111], Python [141], and more recently Kotlin [83] and Dart [68], can be compiled to WebAssembly.

Host environment WebAssembly modules are executed in 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 provided by the JavaScript engine, such as V8 or SpiderMonkey. WebAssembly exports can be wrapped in JavaScript functions using the WebAssembly-JavaScript API [118], allowing them to be called from JavaScript code. Similarly, WebAssembly code can import and call JavaScript functions. Other host environments for WebAssembly include server-side environments like Node.js [127] and stand-alone VMs like Wasmer [178], which provide their own APIs for WebAssembly modules to use. For instance, modules running on stand-alone VMs may interact with the local file system through WASI [177].

V8 compilation The V8 engine, which is used in Google Chrome, employs a two-tiered WebAssembly compilation pipeline [78]. 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 optimisations. 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 optimising TurboFan compiler. Unlike Liftoff, TurboFan is a multi-pass compiler that constructs multiple internal representations of the code during compilation, enabling advanced optimisations. When a function is deemed hot, TurboFan is triggered to recompile and optimise it in the background. The resulting optimised native code then replaces the existing Liftoff-generated code, ensuring increased performance for all subsequent calls to that function.

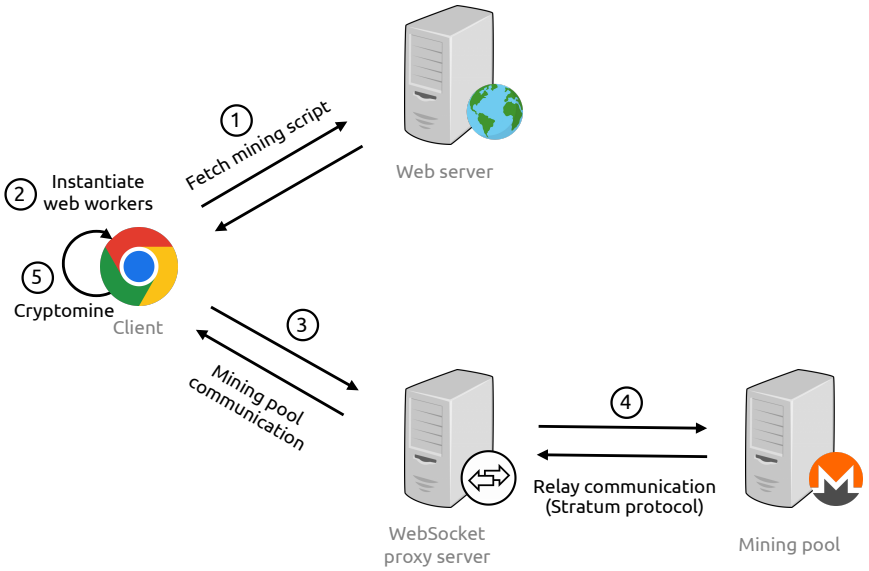


Figure 2.3: Drive-by mining process: The mining script is fetched from the web server, which instantiates the web workers and connects to the WebSocket proxy server, which relays the communication back to the mining pool.

2.4 Drive-by mining

Drive-by mining[4] 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 [114] and VerusCoin [169], which can be mined using standard consumer-grade CPUs [113]. Drive-by mining can be executed through self-hosted mining or by compromising web servers through software vulnerabilities [9, 97, 134, 166] or misconfigurations [130], and then subsequently installing the mining scripts on the compromised web servers. Alternatively, mining scripts can be distributed through advertising platforms [74, 28], compromised third-party libraries integrated within various websites [182], or through adversarial Docker images [31]. Interestingly, browser-based crypto mining has been used as an alternative income stream by organisations like UNICEF [98], although with user approval, differentiating it from adversarial drive-by mining.

Drive-by mining was initially implemented with JavaScript only and popularised with the launch of CoinHive in 2017 [26]. However, 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 drive-by mining incidents in 2018 [2]. By 2019, more than half of all websites containing WebAssembly used it for drive-by mining [120]. Although CoinHive shut down in 2019 [138],

[4] Also referred to as cryptojacking.

reports from the first three quarters of 2022 confirm the steady growth of drive-by mining [94], indicating that it remains a pervasive problem.

Drive-by mining is usually 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 2.3. 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 thread, depending on how many cores are available. 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 [157].

2.5 Analysis techniques

Analysis techniques can be used in a variety of ways, not only for detecting malware but also for vulnerability identification [156, 104], performance optimisation [164], and for understanding and debugging code [125]. Although different in their objective, these techniques often follow similar methodologies. As such, these analysis techniques, including those evaluated in this thesis, can be classified over the following dimensions:

Static and dynamic Static analysis evaluates the program without executing it, enabling fast, real-time detection. Its precision, however, often falls short of dynamic analysis due to its reliance on approximating the actual runtime behaviour [55]. Obfuscation has also been found to be effective against static analysis [116, 21]. Conversely, dynamic analysis observes the behaviour of the program during execution, typically providing higher accuracy than static analysis [55]. Although advantageous, it can be resource-intensive and time-consuming, and it can struggle to explore all possible program behaviours due to a large or potentially infinite search space [183]. While generally more resilient to obfuscation, the presence of obfuscation can increase time consumption and lead to a considerable false positive rate [15, 100].

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 that the rules are precisely formulated. However, their capability can be undermined by novel threats or obfuscation techniques that sidestep established rules [115, 34]. On the other end of the spectrum, machine learning models are trained on large datasets, where each entry is annotated with the expected output. Following this training, the models apply the learned patterns to predict outcomes on previously unobserved instances. Although machine learning methods generally handle obfuscation better than rule-based methods, they are still susceptible to adversarial attacks. Adversarial attacks involve making subtle modifications to input data with the intent to deceive the model, thereby reducing its accuracy [160, 76, 87].

2.5.1 Detecting drive-by mining

To address the ever-increasing threat of drive-by mining, a number of analysis techniques have been proposed. The literature presents methods based on both static [122, 146, 95, 145] and dynamic analysis [176, 143, 93, 92, 18], using both rule-based [146, 95, 176, 18] and machine learning-based [122, 145, 143, 93, 92] approaches. Static methods use a wide range of techniques, including signature matching [95], control flow graph (CFG) analysis [146], and neural networks [122, 145]. Dynamic methods rely on behavioral characteristics such as processor [143, 18] and memory [143] usage, network traffic [143, 93, 92], and JavaScript events [143, 93, 92].

In this thesis, we focus on static analysis techniques for several reasons. Primarily, dynamic analysis proves impractical due to its substantial overhead, ranging from 40% to 100% [143, 176]. 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 [7, 25]. Moreover, several dynamic methods depend on platform or browser-centric features, such as the MessageLoop event [93], or Chrome debugging features [143], 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 drive-by mining, with F_1 scores ranging from 0.95 [122] to 1.00 [146], compared to dynamic methods scoring between 0.96 [92] and 0.99 [18].

MINOS

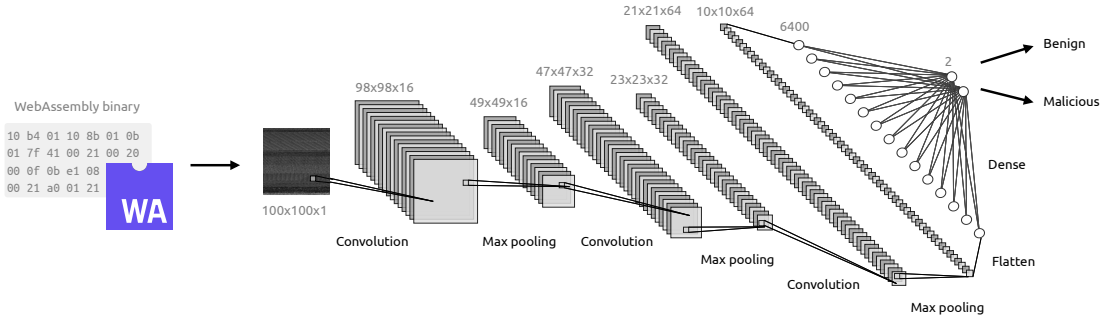


Figure 2.4: Overview of MINOS: The WebAssembly binary is converted to a grayscale image and fed to the MINOS network. The network predicts whether the binary is benign or malicious.

MINOS is a machine learning-based method that uses an image-based classification deep learning approach to identify drive-by mining [122]. As illustrated in Figure 2.4, 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 drive-by mining 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.

WASim

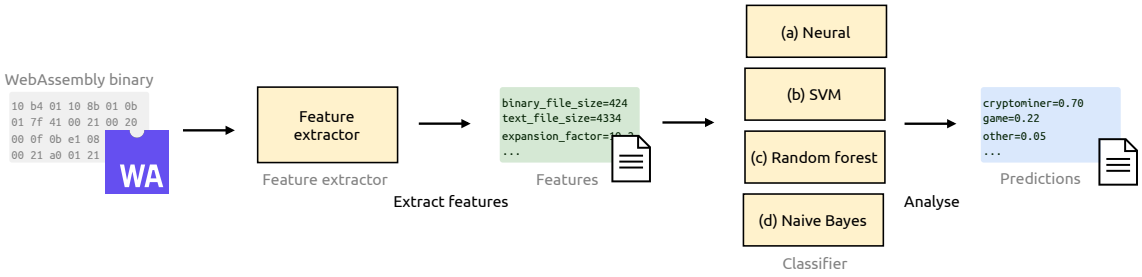


Figure 2.5: Overview of WASim: Features are extracted from the WebAssembly binaries and fed into a classifier. The classifier model is either: (a) Neural, (b) SVM, (c) Random forest, or (d) Naive Bayes. The classifier outputs a usage report containing the predictions.

WASim is a machine learning-based method that extracts and analyses a set of features for detecting drive-by mining [145]. The procedure is depicted in Figure 2.5. First, the WebAssembly binary is converted from the binary format (`wasm`) to the text format (`wat`), which is then parsed to extract the features listed in Table 2.2. 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.

MinerRay

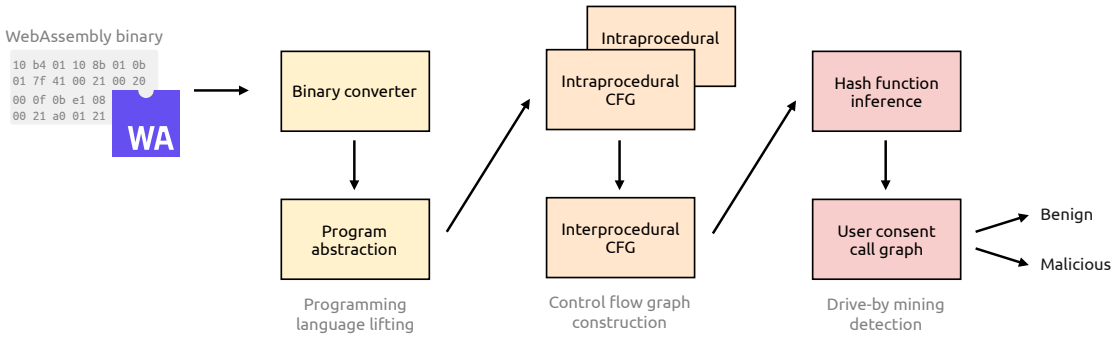


Figure 2.6: Overview of MinerRay: The WebAssembly binary is converted into a custom intermediate language, from which an interprocedural CFG is constructed. The control flow is then analysed to detect drive-by mining, as well as for checking user consent.

MinerRay is a rule-based method that analyses the control flow of the WebAssembly module to detect drive-by mining [146]. As illustrated in Figure 2.6, the process is divided into three parts:

Category	Feature
File related	Binary file size
	Text file size
	Expansion factor
Program complexity	Total lines of code
	Minimum lines of code in functions
	Maximum lines of code in functions
	Average lines of code in functions
	Number of functions
WebAssembly specific	Is an asm.js module
	Number of types
	Number of data sections
	Number of import functions
	Number of export functions
Function signature	Names of import functions
	Names of export functions

Table 2.2: Features extracted by WASim that are subsequently used for classification.

Programming language lifting, CFG construction, and drive-by mining 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 analysing 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 uses an extensive set of antivirus scanners to detect malware [170]. 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 drive-by mining [84, 176, 25].

2.6 Obfuscation

Obfuscation involves the process of transforming a given program into one that is syntactically different but semantically equivalent [121]. It has been used for a variety of purposes, including prevention of reverse engineering [32, 154, 20], prevention of software modification [10, 72], hiding static data [96, 91], and for malware evasion [115, 133, 107]. 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 [41].

The transformations applied in this thesis, aligned with the taxonomy proposed by Collberg et al. [41], are categorised as follows:

Control obfuscation Control obfuscation manipulates the control flow of the program. Transformations can affect control *aggregation* by splitting or merging computations [86, 82], control *ordering* by randomising computation sequences [3, 89], and control *computations* by inserting redundant or dead code [39, 174], or by incorporating opaque predicates [42, 148]. Such techniques can include false or irrelevant conditional statements, embedding loops within loops, altering loop conditions, flattening the control flow, or replacing control structures with complicated equivalents. These transformations can significantly impede both static and dynamic analysis efforts, making the deciphering of the actual execution flow of the program a challenging task [108, 42]. However, for transformations that alter the control flow, a certain amount of computational overhead is usually unavoidable [41].

Data obfuscation Data obfuscation is a technique that transforms the representation of data structures and values within the code while preserving their semantic meaning. Such techniques can include splitting of data structures [58, 155], using complex encodings or encryption [165, 71], or renaming variables and functions with obscure names [57].

Preventive transformations Preventive transformations are designed to thwart certain types of code analysis or reverse engineering techniques. They typically target specific characteristics or behaviours of known analysis tools, creating code that is resistant to those particular methods. Such transformations can include techniques like anti-debugging (preventing the code from being run in a debugger) [29], anti-tampering (making it hard to modify the code) [139], or encoding mechanisms that resist automatic de-obfuscation.

Layout obfuscation Layout obfuscation involves modifying the arrangement of code elements. Transformations may include removal of source code formatting [32, 5], reordering instruction sequences [39], or modifying the arrangement of functions and variables [27, 148]. The objective is to disrupt the readability and traceability of the code, thus deterring reverse engineering attempts.

2.6.1 Diversification

Diversification is a technique used to generate multiple distinct yet semantically equivalent versions of a program. Unlike obfuscation, which is primarily concerned with making a program difficult to understand or analyse, diversification focuses on creating a multitude of program versions to improve resilience against attacks [39, 69]. This strategy proves particularly beneficial when a group of software systems are under attack; different variants ensure that not every instance is vulnerable to the same exploit [85].

Although both diversification and obfuscation produce different variants of programs, their objectives and methods differ significantly. Obfuscation makes the code harder to understand or analyse, typically without regard to how many variants are produced. Diversification, on the other hand, deliberately creates multiple variants to enhance security without necessarily aiming to obfuscate the code. Nevertheless, it is important to note that diversification can inadvertently obfuscate the code, thereby evading malware detection [24, 135]. Given this, while *mutations* is a term often used by diversifiers, within this discussion, a *transformation* will serve as an interchangeable term to denote either an obfuscating transformation or a diversifying mutation.

2.6.2 Obfuscation tools

Throughout the years, a multitude of obfuscation tools have been developed for a variety of programming languages and architectures. For instance, Microsoft's portable executable (PE) format can be obfuscated using tools like Themida [132], VMProtect [171], ExeCryptor [64], Code Virtualizer [162], and Enigma Protector [140]. Similarly, x86 binaries can be obfuscated using tools such as objobf [129], xObf [54], and x86-Code-Virtualizer [126].

Options for WebAssembly obfuscation are comparatively limited. The only known tool that operates at the WebAssembly level is the wasm-mutate diversifier [23]. Nevertheless, a viable alternative 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. A variety of tools are available for C code, including Tigress [40], Stunnix [158], CodeMorph [38], CShroud [53], and COBF [37]. Only Tigress and Stunnix are actively maintained; however, Stunnix requires a license. LLVM bitcode can be obfuscated using tools such as obfuscator-LLVM (OLLVM) [90].

In this thesis, the following obfuscation tools are used:

Tigress Tigress [40] is a source-to-source obfuscator written in OCaml. It transforms a C99 standard-compliant C source file into an obfuscated equivalent. It provides a wide range of features, including virtualisation with a randomly generated instruction set architecture (ISA), control flow flattening, function splitting and merging, data encoding, and preventive transformations like anti-taint analysis and anti-alias analysis. Tigress has been thoroughly evaluated in the literature [17, 12, 152], and has been found to be successful in evading drive-by mining detection [17].

OLLVM OLLVM [90] is implemented as middle-end passes in the LLVM compilation suite. Transformations include control flow flattening, bogus control flow, basic block splitting, string encryption, and instruction substitution, to name a few. It is almost completely language and platform-independent, working with all programming languages that are supported by LLVM, notably C, C++, and Objective C. It has been used for preventing reverse engineering [12, 99], as well as for evading malware detection [136].

wasm-mutate wasm-mutate [23] is a wasm-to-wasm diversifier written in Rust. It takes a WebAssembly module as input and returns a mutated variant of it. It offers 135 mutations, categorised into peephole, control flow, and module structure mutations. Wasm-mutate has been used for fuzzing [6] and has been found to be successful in evading drive-by mining detection [25].

2.6.3 Obfuscation for WebAssembly

WebAssembly obfuscation, a relatively underexplored field, has seen some intriguing investigations lately. In their WiSec '22 contribution, Bhansali et al. [17] used Tigress to obfuscate C source code before compiling it to WebAssembly. Although they successfully evaded MINOS detection, they did not ensure the WebAssembly binaries were functional after obfuscation, nor did they measure the overhead caused by obfuscation. Further, an impending study in Computers & Security '23 by Cabrera Arteaga et al. [25] shows how wasm-mutate can be used to diversify WebAssembly binaries, effectively evading both MINOS and VirusTotal, with minimal performance overhead. Loose et al. [103] have proposed a forthcoming novel technique in DIMVA '23, employing binary manipulation through instrumentation to incorporate adversarial examples into code sections within the module, successfully evading MINOS. However, they measured the performance overhead using a generic SHA256-hashing library instead of crypto mining binaries, which threatens the validity of their results.

In a similar vein, studies have turned to WebAssembly as a means of obfuscating JavaScript. Romano et al. [144] 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. [175] introduced JSPro, a tool that also 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.

2.7 Code similarity

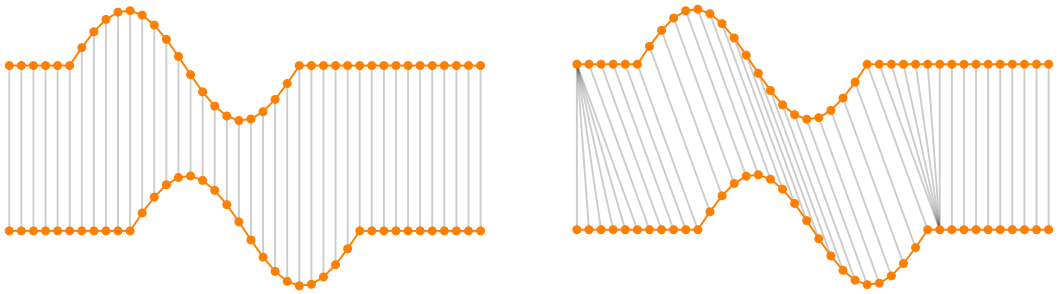
Traditional approaches for evaluating code similarity have typically relied on complexity metrics such as cyclomatic complexity [110], Halstead complexity measures [81], and lines of code. These metrics focus on the structural complexity of the code in terms of the control flow with cyclomatic complexity, the usage of operators and operands with Halstead complexity, and the overall size of the code with the lines of code metric.

Although these metrics have been widely used, they have several limitations [128, 131]. Their primary shortcoming is their high-level operation, which fails to identify the detailed semantics of the code. For instance, code segments with the same structure but different functionality would be considered similar under these metrics. Moreover, they are sensitive to minor transformations, like code reordering, renaming, or refactoring, that maintain functionality while modifying the surface structure of the code.

More importantly, these conventional complexity metrics fall short when it comes to assessing the similarity between WebAssembly binaries. Since WebAssembly is a stack-based language, each operation is executed by pushing to and popping values from the stack. The sequence in which these operations are carried out affects the semantics of the program. For instance, a simple inversion of the push and pop operations sequence can either drastically change the functionality of a code segment or render it entirely non-functional. Traditional complexity measures do not account for the order of instructions and are therefore not suitable for WebAssembly.

2.7.1 Sequence alignment

Sequence alignment methods take into account the order of instructions and have therefore been employed in this thesis. Several distance-based sequence alignment methods have been proposed in the literature, such as the Euclidean distance [48], longest common sub-sequence [4], and dynamic time warping (DTW) [47]. Each of these methods offers unique strengths and is capable of providing more granular control over complexity metrics. However, DTW has proven to be more accurate than other sequence alignment algorithms [1, 13], like the Euclidean distance, which can be visually seen in Figure 2.7.



(**a**) Euclidean distance.

(**b**) Dynamic time warping.

Figure 2.7: Comparison of the Euclidean distance and DTW for sequence alignment. The Euclidean distance is calculated as the sum of the absolute differences between the corresponding elements of the two sequences. The DTW distance is calculated as the sum of the absolute differences between the corresponding elements of the two sequences, where the elements are aligned in such a way that the sum is minimised. The figure is a modified version from Tavenard et al. [161].

Dynamic time warping

DTW is a method used to measure similarity between two temporal sequences, which may vary in time or speed [119]. Initially designed to address problems in speech recognition [149], it was subsequently introduced to the field of time series by Berndt et al. [16]. DTW has demonstrated its versatility by effectively addressing a myriad of challenges across a wide range of domains, such as robotics, biometrics, and meteorology [119]. Notably, it has also been successfully used for aligning and comparing stack traces [106, 8] and WebAssembly binaries [7].

The fundamental concept of DTW, as shown in Figure 2.7b, is to identify the optimal alignment between two sequences. The process aims to minimise 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 minimises the total cumulative distance. Notably, DTW can effectively compare sequences of varying lengths, negating the need for normalisation [142].

A particularly useful feature of DTW is that it can process any data that can be represented as a sequence. Leveraging this, we can represent WebAssembly binaries as sequences of instructions and compare them using DTW. DTW finds the distance between the two WebAssembly binaries by aligning one sequence of instructions with the other by "warping" the time dimension of one or both sequences. This distance serves as a dissimilarity metric between the binaries, with larger distances indicating substantial dissimilarity.

However, traditional DTW algorithms can cause substantial computational overheads, particularly for lengthy sequences. This is due to the necessity of calculating and storing the entire cost matrix, which incurs a time complexity of $\mathcal{O}(N^2)$. To avoid this, FastDTW was introduced as an approximation of DTW that operates in near-linear time complexity with comparable accuracy [150].

Chapter 3

Methodology

This chapter presents the methodology used in this thesis. The experimental setup, including the system configuration and dataset used, is described in Section 3.1. Then, Section 3.2 provides an in-depth overview of the implementation of obfuscation, de-obfuscation, drive-by mining detection, how we extracted the native code, measuring the hash rate, and DTW. Lastly, Section 3.3 presents the metrics used to evaluate the data collected.

In the interest of transparency and reproducibility, the code used to conduct the experiments is publicly available on GitHub.[1]

3.1 Experimental setup

3.1.1 System configuration

The experiments were performed on a VM provided by the Norwegian University of Science and Technology (NTNU). Note that although the VM has 64 processor cores available, the experiments relating to drive-by mining were performed using only 4 cores, making the results comparable with consumer-grade hardware and mobile devices. Moreover, the code has been containerised using Docker to ensure that the experiments are reproducible across several system configurations. Table 3.1 presents a detailed specification of the system.

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

Component	Specification
Operating system	Debian 10
Kernel	4.19.0-22
Processor	AMD EPYC 7742
Processor frequency	2.24 GHz
Processor cores	64 cores
L1 cache	32K
L2 cache	512K
L3 cache	16384K
Physical memory	64 GB

Table 3.1: System specification.

3.1.2 Dataset

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 3.2: Dataset used in this study, spanning a wide range of categories, including utilities, games, and crypto miners.

Detailed in Table 3.2, the dataset used in this thesis covers a broad spectrum of categories, including utilities, games, and crypto miners. All of these 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. Meanwhile, the games are chosen to represent a wide range of complexity, ranging from classical arcade games to cellular automaton simulations. Lastly, the crypto miners contain the predominant versions of the CryptoNight algorithm, as well as their less memory and computation-intensive variants. This should provide extensive coverage of crypto mining malware, as several studies have found that in-the-wild drive-by mining implementations are all based on the CryptoNight algorithm [25, 84].

3.2 Implementation

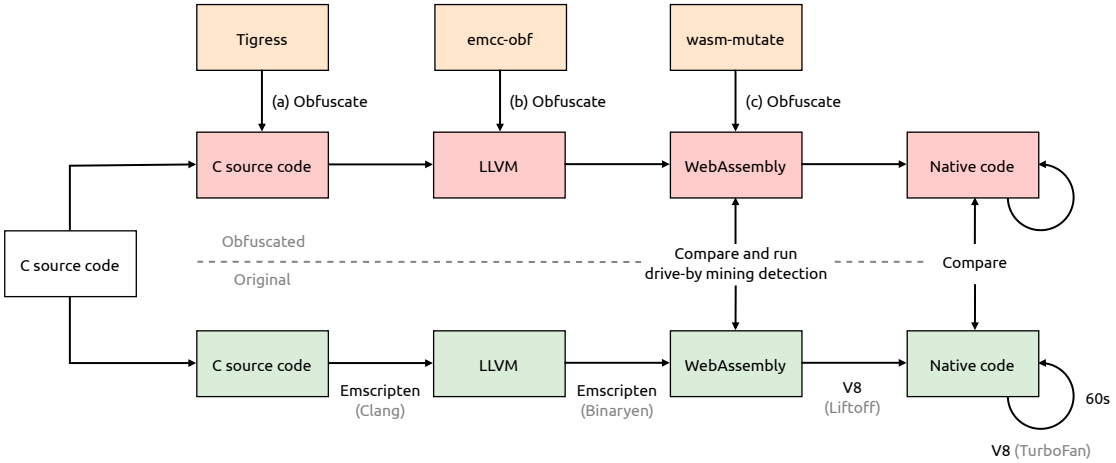


Figure 3.1: Overview of the research strategy: Each program is obfuscated using either (a) Tigress, (b) emcc-obf, or (c) wasm-mutate at the corresponding abstraction level, and finally compiled to WebAssembly. The WebAssembly binaries are then run through the drive-by mining detectors, instantiated in the browser to extract the native code, and compared with their non-obfuscated counterparts.

The implementation, aligned with the research strategy depicted in Figure 3.1, is described in the following sections. Each application in the dataset undergoes obfuscation using Tigress, emcc-obf, and wasm-mutate, as specified in Section 3.2.1. After obfuscation, the WebAssembly binaries are attempted de-obfuscated through compiler optimisations, described in Section 3.2.2, before passing the original, obfuscated, and de-obfuscated binaries to the drive-by mining detectors, detailed in Section 3.2.3. Moreover, the WebAssembly binaries are instantiated within the Chrome browser to extract the native code, as explained in Section 3.2.4. To measure the performance overhead of the crypto miners, we implement a drive-by-mining client, web server, and WebSocket proxy server, described in Section 3.2.5. Lastly, we implement the DTW algorithm as a means of comparing the WebAssembly binaries, detailed in Section 3.2.6.

3.2.1 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 3.2 are obfuscated using Tigress, emcc-obf, and wasm-mutate. The number of binaries generated for each application is shown in Table 3.3. For Tigress and emcc-obf, we apply eight transformations per application, while for wasm-mutate, we apply six transformations per

Category	Original	Tigress	emcc-obf	wasm-mutate	stacked wasm-mutate
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 3.3: Number of WebAssembly binaries in the original and obfuscated case.

application. Since wasm-mutate produced unsatisfactory results when applying individual transformations, and in an effort to replicate results from other studies [25], we also stack the wasm-mutate transformations. To this end, we apply 1000 random stacked transformations to each application in the dataset. This 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) [124]. In this process, we identified[2] and fixed[3] a bug in CIL, enabling us to process all the applications in our dataset. Although only Tigress requires a single source file, we use the same 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 WebAssembly binaries using the same Emscripten version. The applications were compiled with optimisation turned off to ensure the compiler did not optimise away the obfuscation applied.

To ensure correctness, we invoke all the binaries in the browser and manually check that they are still functioning as intended. For the stacked wasm-mutate transformations, we check every 100th iteration. For the crypto miners, we ensure that the hashes reach the mining pool and are accepted by it as valid hashes. We were only able to verify this for cn-r, cn-lite-0, and cn-pico-trtl, due to the mining network difficulty, which led to the receipt of new jobs before the other variants could solve and submit the hashes. Thus, for all CryptoNight variants, we compare the first 100 hashes of the original and obfuscated binaries and ensure that they are identical. We found that the applications in our dataset still functioned as intended after obfuscation.

Tigress

We use the latest version of Tigress, version 3.3.2, to obfuscate the source code of each application. To that end, we apply the following transformations 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.

[2] https://github.com/goblint/cil/issues/137
[3] https://github.com/goblint/cil/pull/138

- **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.
- **Virtualisation:** Control obfuscation that transforms a function into a specialised interpreter by constructing a unique bytecode. This technique involves the creation of a virtual 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) [65].
- **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.
- **Ant-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.

emcc-obf

We build and release emcc-obf,[4] the first WebAssembly compiler with built-in obfuscation support. It is a modified version of the Emscripten compiler, one of the most popular WebAssembly compilers. The modifications are based on OLLVM [90], which is no longer maintained. Instead, we use the Hikari obfuscator,[5] 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[6] and Emscripten[7] versions. The precise commit hashes were found using the Emscripten releases page.[8]

We apply the following transformations 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.

[4] https://github.com/HakonHarnes/emcc-obf
[5] https://github.com/61bcdefg/Hikari-LLVM15
[6] Commit hash: ecbebfbee12f2f25af648119604915fc37427f6f
[7] Commit hash: fab93a2bff6273c882b0c7fb7b54eccc37276e03
[8] https://chromium.googlesource.com/emscripten-releases/

- **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.

wasm-mutate

We use wasm-tools[9] 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. Using wasm-mutate, we apply the following transformations to the WebAssembly binaries of each program:

- **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, localised 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.

[9] https://github.com/bytecodealliance/wasm-tools

In addition to applying the individual transformations, we also stack them. To this end, we apply 1000 random stacked transformations to each application. This is similar to the "Baseline evasion" algorithm proposed by Cabrera et al. [25].

3.2.2 De-obfuscation

To our knowledge, there are no publicly available de-obfuscation tools for WebAssembly. As a substitute, we use compiler optimisations to automatically de-obfuscate the WebAssembly binaries. We use `wasm-opt` from Binaryen, which can be applied directly to the WebAssembly binaries. We apply the highest optimisation level, `-O3`, with the aim of reversing as many transformations as possible.

We optimise all of the binaries in the dataset, including the benign ones, so we have a point of comparison. In total, we optimise 22 506 binaries, leaving us with 22 506 optimised binaries and 22 506 non-optimised binaries, for a total of 45 012 binaries evaluated in this thesis.

3.2.3 Drive-by mining detection

In order to identify drive-by mining, we implement the detection methods presented in the background chapter; namely MINOS, WASim, MinerRay, and VirusTotal. For MINOS, we use the reproduction by Cabrera et al.[10] We use the publicly available implementation for WASim,[11] although we encountered challenges due to out-of-date dependencies, which we then updated to their latest versions.[12] We also use the public implementation of MinerRay,[13] albeit faced with issues related to the JavaScript heap limit due to the path explosion problem, which led us to disable the function call linking for larger files as advised by the authors.[14] This action, however, may affect the detection rate. Additionally, we implemented a two-minute timeout delay to prevent indefinite execution. For VirusTotal, we use their API.[15]

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.

3.2.4 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

[10] https://github.com/ASSERT-KTH/wasm_evasion/tree/main/malware_reproduction/rq2
[11] https://github.com/WASimilarity/WASim
[12] https://github.com/WASimilarity/WASim/pull/20
[13] https://github.com/miner-ray/miner-ray.github.io
[14] https://github.com/miner-ray/miner-ray.github.io/issues/20
[15] https://developers.virustotal.com/reference/overview

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 optimisation. 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.

3.2.5 Measuring the hash rate

To measure the hash rate, we implement a drive-by mining client, web server, and WebSocket proxy server as described in Section 2.4. We use the public Webminerpool[16] 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 containerised the client and server, as well as disabled caching, to ensure that the environment was consistent across all experiments.

3.2.6 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 begin by converting 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. To implement this, we use the `fastdtw`[17] python library.

3.3 Evaluation metrics

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

3.3.1 RQ1 – Effectiveness

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

Definition 1 (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.

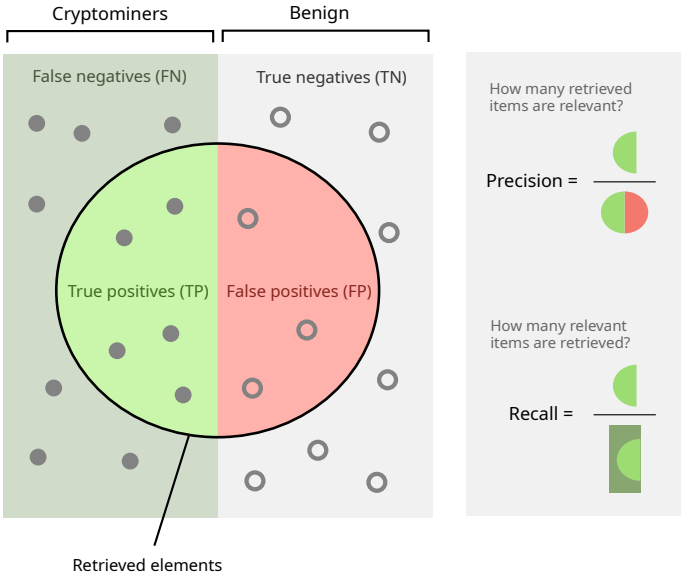
[16] https://github.com/notgiven688/webminerpool
[17] https://pypi.org/project/fastdtw

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 3.2.4 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 2 (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. Specifically, if N denotes the size of the original native code, and N' is the size after obfuscation, then:

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

3.3.2 RQ2 – Detectability



Abbreviations: True positive (TP), True negative (TN), False positive (FP), False negative (FN).

Figure 3.2: Precision is how many retrieved items are relevant, while recall is how many relevant items are retrieved. The figure shown is a modified version from Wikipedia [52].

To assess how effective the obfuscation methods are in evading detection, we feed the obfuscated binaries to the drive-by mining detectors presented in Section 3.2.3. Following this, we calculate the resulting precision, recall, and F_1 scores to assess the accuracy of the detection methods. These measures, illustrated in Figure 3.2, are formally defined as follows:

Definition 3 (Precision)**.** Precision measures how many of the retrieved items are relevant. In the context of drive-by mining, it measures how many of the items identified as crypto miners are actually crypto miners. Precision is formally defined as:

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

Definition 4 (Recall)**.** Recall measures how many of the relevant items are retrieved. In the context of drive-by mining, it measures how many of the crypto miners are identified as crypto miners. Recall is formally defined as:

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

Definition 5 (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 calculated as:

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

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

3.3.3 RQ3 – Reversibility

To evaluate the resilience of the applied transformations, we de-obfuscate the WebAssembly binaries as described in Section 3.2.2. Then, we compute the distances (Definition 1) of the de-obfuscated binaries, comparing them with the distances of the obfuscated binaries. A reduction in distance following de-obfuscation indicates the successful removal of obfuscation and, thereby, a successful de-obfuscation process.

3.3.4 RQ4 – 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 6 (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:

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

For the crypto mining binaries, we measure and compare the hash rates in the original and obfuscated cases. To this end, we implement a drive-by mining setup as described in Section 3.2.3. We let the binaries calculate hashes for 100 seconds before measuring the total hashes.

Definition 7 (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 8 (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:

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

Chapter 4

Results

This chapter presents the results of the experiments described in Chapter 3. These results are systematically structured and presented in alignment with the research questions.

In Section 4.1, we start by evaluating the effectiveness of obfuscation through the analysis of the WebAssembly binaries and the resulting native code after compilation in the browser. Then, in Section 4.2, we examine if obfuscation can evade the drive-by mining detectors. Moving forward, we analyse the reversibility of the transformations by performing automatic de-obfuscation in Section 4.3. Lastly, we measure the overhead introduced by the transformations by analysing the file size and hash rates in Section 4.4.

4.1 Effectiveness

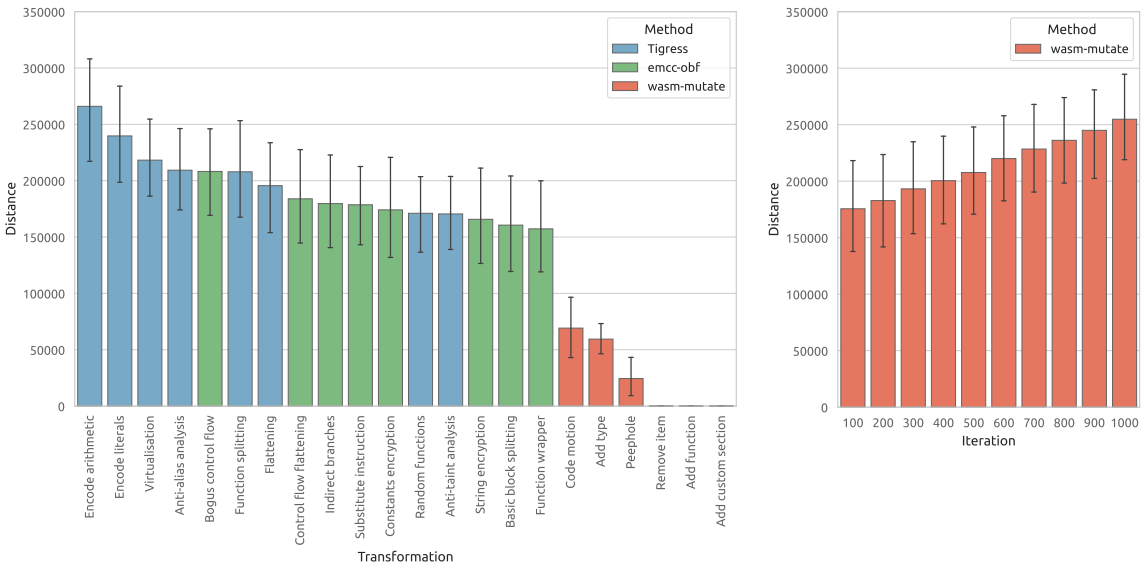
The objective of this section is to address the following research question:

RQ1 – Effectiveness How effective are the transformations at obfuscating WebAssembly, and how is the resulting native code affected?

4.1.1 Distances after obfuscation

Figure 4.1 shows the distances (Definition 1) between the original and obfuscated WebAssembly binaries for each obfuscation method, transformation, and iteration. Larger distances suggest more dissimilar binaries, indicating more effective obfuscation.

Encode arithmetic, encode literals, and virtualisation are the most effective transformations, which were all applied using Tigress. In fact, Tigress is the most effective obfuscation method, with an average distance of 209k, followed by emcc-obf and wasm-mutate, averaging 176k and 30k, respectively. Although individual transformations are less effective for wasm-mutate, stacked transformations yield significant distances, reaching a distance of 252k, rivalling the effectiveness of Tigress.



(**a**) Distances for each transformation.

(**b**) Distances for each iteration.

Figure 4.1: Distances for each obfuscation method, transformation, and iteration sorted in descending order. The error bars shown are indicative of a 95% confidence interval.

Interestingly, there are differences in which obfuscation type is the most effective for each obfuscation method. For Tigress, data obfuscations such as encode arithmetic and encode literals outperform control obfuscations like virtualisation and function splitting. In contrast, control obfuscations like control flow flattening and code motion are more effective than data obfuscations like constants encryption and peephole for emcc-obf and wasm-mutate.

Figure 4.2 shows the distances (Definition 1) for each obfuscation method and transformation 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. 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.

There are notable differences between the various application categories. 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, virtualisation 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 to be obfuscated.

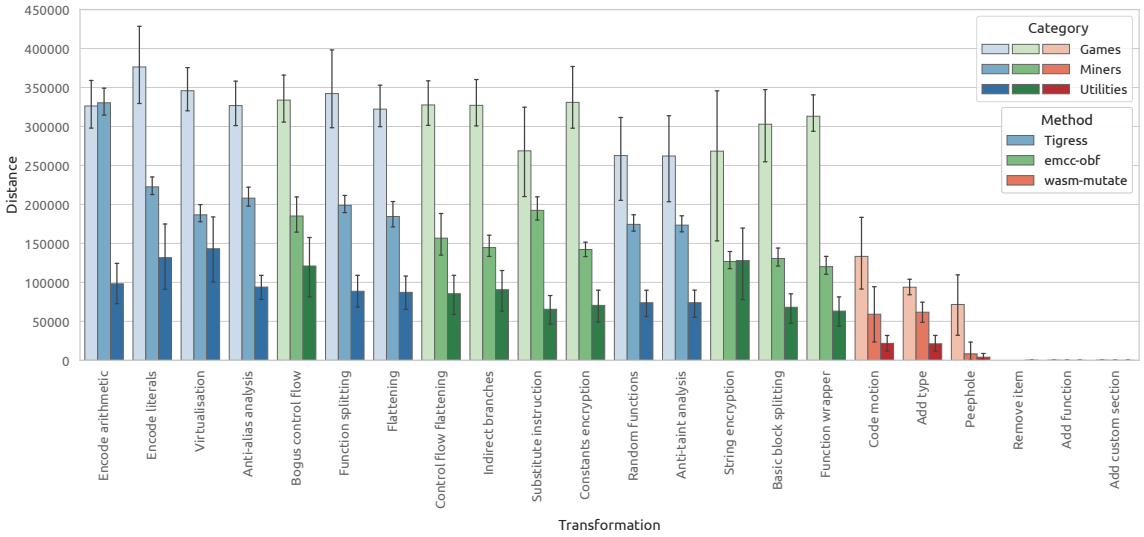


Figure 4.2: Distances for each obfuscation method and transformation grouped by program category, sorted by the average distances in descending order. The error bars shown are indicative of a 95% confidence interval.

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 underscores the significant influence the content of the application has on the effectiveness of the transformations, irrespective of the abstraction level at which they are applied.

4.1.2 Native code size increase

Figure 4.3 presents the relative increase in native code size (Definition 2) for each obfuscation method, transformation, and iteration after undergoing lazy compilation (Liftoff) and optimisation (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.

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 instance, consider a situation where Liftoff initially generates 100MB of native code. TurboFan optimises this to 50MB. After obfuscation, Liftoff's output increases to 200MB, and TurboFan's optimisation reduces this to 150MB. So, despite Liftoff showing a 100% relative increase and TurboFan a 200% relative increase after obfuscation, TurboFan's optimisation still results in an overall reduction of native code.

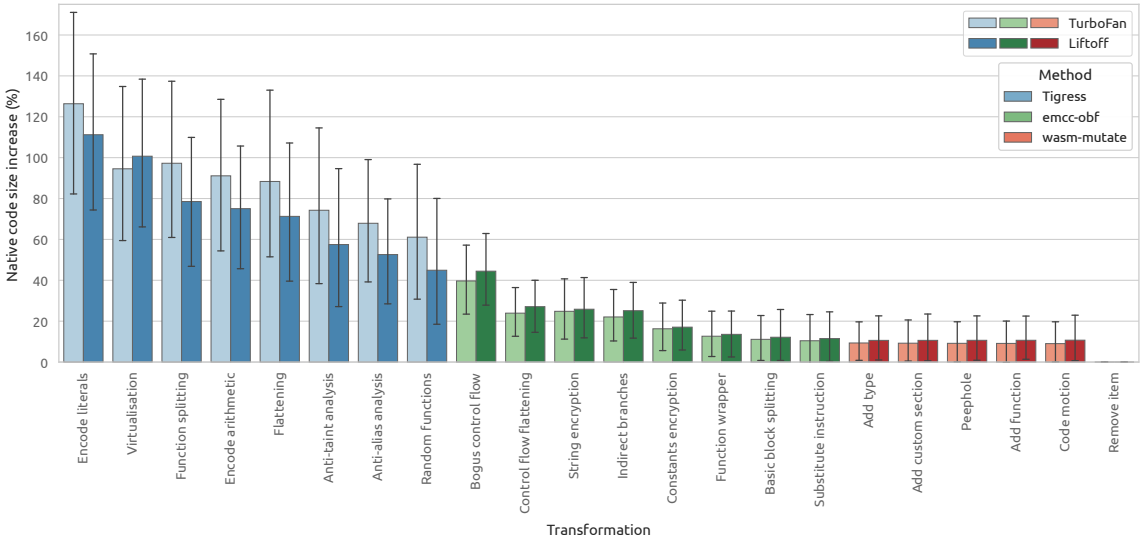


Figure 4.3: Native code size increase for each obfuscation method and transformation after lazy compilation (Liftoff) and optimisation (TurboFan) in the V8 engine, sorted by the average native code size increase in descending order. The error bars shown are indicative of a 95% confidence interval.

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. Comparatively, emcc-obf causes considerably smaller increases with an average of 20% and 22% after Liftoff and TurboFan compilation, respectively. On the contrary, wasm-mutate demonstrates the slightest increase of 10% for both Liftoff and TurboFan. However, when stacking the transformations, wasm-mutate substantially increases the native code size, with relative increases ranging from 16% to 140%, as can be seen in Figure 4.4.

There are differences in which types of transformation induce the largest native code for each obfuscation method. For Tigress, data obfuscation, such as encode literals, causes a more substantial increase in native code size than control obfuscations like virtualisation. However, for emcc-obf, control obfuscations such as bogus control flow and control flow flattening impose a larger increase in native code compared to data obfuscations like constants encryption and string encryption. For wasm-mutate, there are no discernible differences between control and data obfuscation.

An intriguing observation is that, for Tigress, the relative increase in native code is larger for Liftoff than for TurboFan. The opposite is true for emcc-obf and wasm-mutate; the relative increase in native code is larger for TurboFan than they are for Liftoff. In either case, Turbofan always decreases the size of the native code, but it is unable to entirely optimise away the extra instructions introduced by obfuscation.

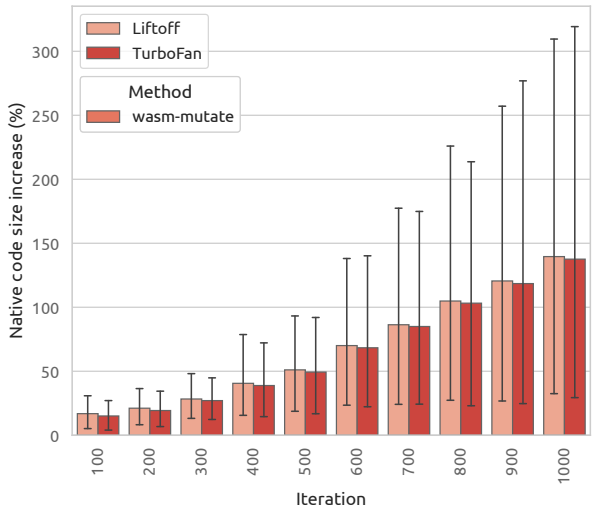


Figure 4.4: Native code size increase for each iteration applied with wasm-mutate after lazy compilation (Liftoff) and after optimisation (TurboFan) in the V8 engine. The error bars shown are indicative of a 95% confidence interval.

Summary The effectiveness of obfuscation depends on the obfuscation method, transformation, and application to be obfuscated. Tigress was slightly more effective than emcc-obf, with wasm-mutate achieving comparable results when stacking the transformations. The most effective transformations included encode arithmetic, encode literals, and virtualisation. While data obfuscation was most effective for Tigress, control obfuscation was most effective for emcc-obf and wasm-mutate. The effectiveness of the transformations depends on the application to be obfuscated, with crypto miners benefitting from arithmetic encoding. Obfuscation consistently increased the native code, with Tigress increasing it the most. Although TurboFan reduced the native code size by 30% on average, it was unable to completely remove the instructions caused by obfuscation.

4.2 Detectability

The objective of this section is to address the following research question:

RQ2 – Detectability How effective is obfuscation at evading state-of-the-art drive-by mining detectors, and which transformations are the most effective?

4.2.1 Detection results

Figure 4.5 shows the F_1 scores (Definition 5) for each detection and obfuscation method. The results are nuanced, with obfuscation sometimes decreasing the accuracy of detection methods, while in other cases increasing it.

Figure 4.5: 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.

We find that detection methods with higher accuracy tend to decrease in accuracy after obfuscation. For instance, 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.

Conversely, 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 4.1 presents the precision (Definition 3), recall (Definition 4), and F_1 scores (Definition 5) of MINOS and WASim (neural) after obfuscation. We exclude WASim (naive) and the other detection methods from the table due to their low accuracy.

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, virtualisation, 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.

		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**
	Virtualisation	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 4.1: Precision, recall, and F_1 scores for MINOS and WASim (neural) after applying obfuscation with Tigress, emcc-obf, and wasm-mutate.

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).

Even after obfuscation, the recall often exceeds the precision for both MINOS and WASim, which is an interesting observation. This suggests that obfuscation is not always effective for evading drive-by mining detection. Instead, the drop in accuracy is primarily due to benign applications being mistakenly identified as crypto miners. There are certainly exceptions to the rule; function splitting and bogus control flow effectively reduce recall for MINOS, and most control obfuscations do the same for WASim.

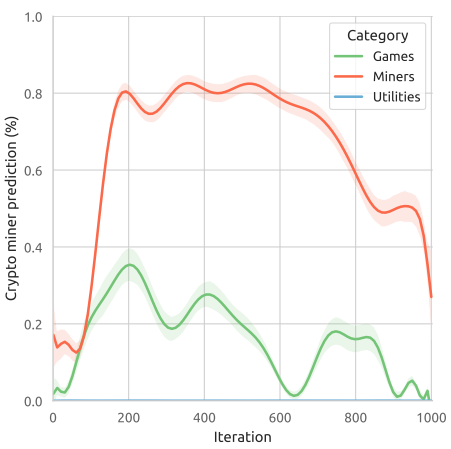
Again, individual transformations are not effective for wasm-mutate. However, stacking the transformations yields promising results. As more transformations are applied, the F_1 score for MINOS consistently decreases. For WASim, however, the F_1 score inconsistently increases from 0.43 to 0.90 after 500 iterations, before decreasing back down to 0.33 after 1000 iterations.

4.2.2 WASim classifiers

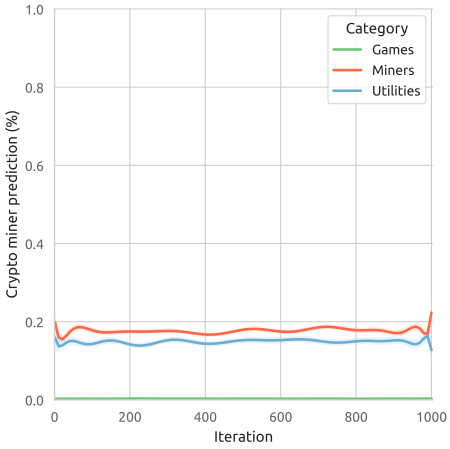
Figure 4.6 shows the predictions of the different WASim classifiers in response to stacked wasm-mutate transformations. Predictions for the naive Bayes, RF, and SVM classifiers remain relatively constant as more transformations are applied. This shows that they are not affected by the transformations, 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. This indicates that the neural classifier is not nearly as obfuscation-resilient as the other classifiers, which is an interesting finding, as the authors of WASim recommend the neural classifier.

This observation encouraged us to investigate which wasm-mutate transformations are the most effective for evading detection. That is, which transformations decrease the crypto mining predictions of WASim (neural) the most? As can be seen in Figure 4.7a, the code motion and peephole transformations decrease the crypto miner predictions the most and are deemed the most effective for evading drive-by mining detection.

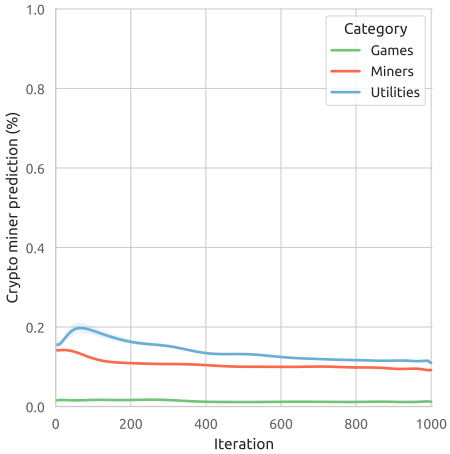
Intrigued by these results, we investigate if we can strategically apply transformations to evade WASim (neural) detection. First, we apply random mutations to the crypto mining binaries and select the resulting crypto mining binaries that have the highest crypto miner predictions. Then, we iteratively apply the code motion and peephole transformations to those binaries and observed the predictions of the WASim (neural) classifier. The results are shown in Figure 4.7b. 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, completely evading detection and demonstrating how WASim (neural) can be strategically evaded.



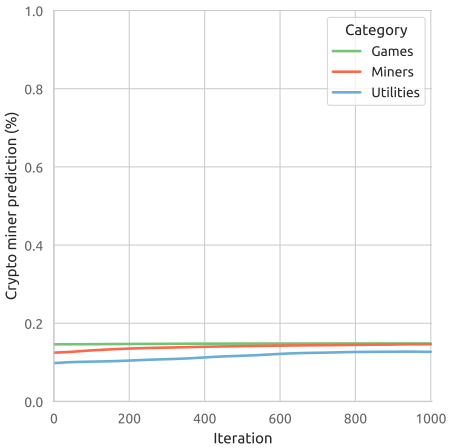
(**a**) Neural.



(**b**) Naive Bayes.

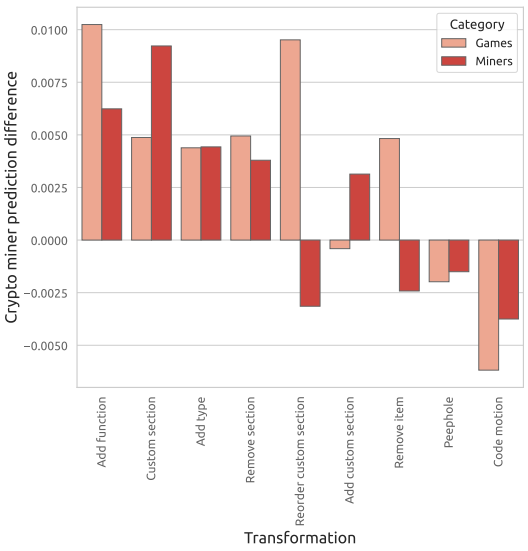


(**c**) RF.

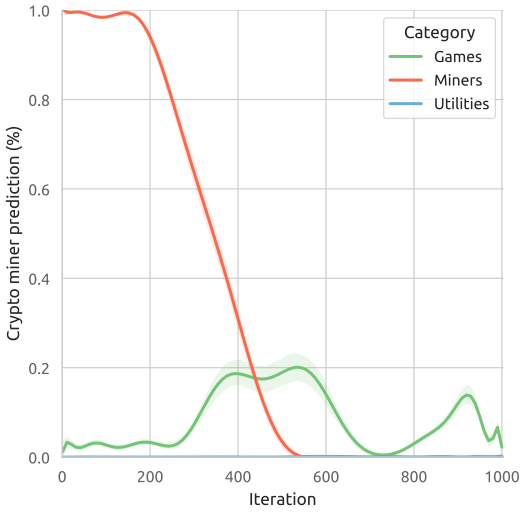


(**d**) SVM.

Figure 4.6: 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.



(**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 4.7: 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.

Summary Obfuscation can be effective at evading state-of-the-art drive-by mining detectors, but it is highly dependent on the specific detection and obfuscation methods, as well as the specific transformation. Tigress proved to be most effective in evading MINOS, while emcc-obf was more effective in evading WASim. Anti-alias analysis was the only transformation that could completely evade detection from both MINOS and WASim. However, for WASim, we found that several control obfuscation transformations were also able to achieve this, and, in general, control obfuscation was more effective than data obfuscation. The decrease in accuracy was often the result of benign applications being mistakenly identified as crypto miners rather than crypto miners evading detection. We also found that wasm-mutate could evade WASim detection completely, but only when strategically applying the most effective transformations; namely code motion and peephole.

4.3 Reversibility

The objective of this section is to address the following research question:

RQ3 – Reversibility How resilient are the transformations to automatic de-obfuscation, and how does de-obfuscation affect the detection accuracy?

4.3.1 Distances after de-obfuscation

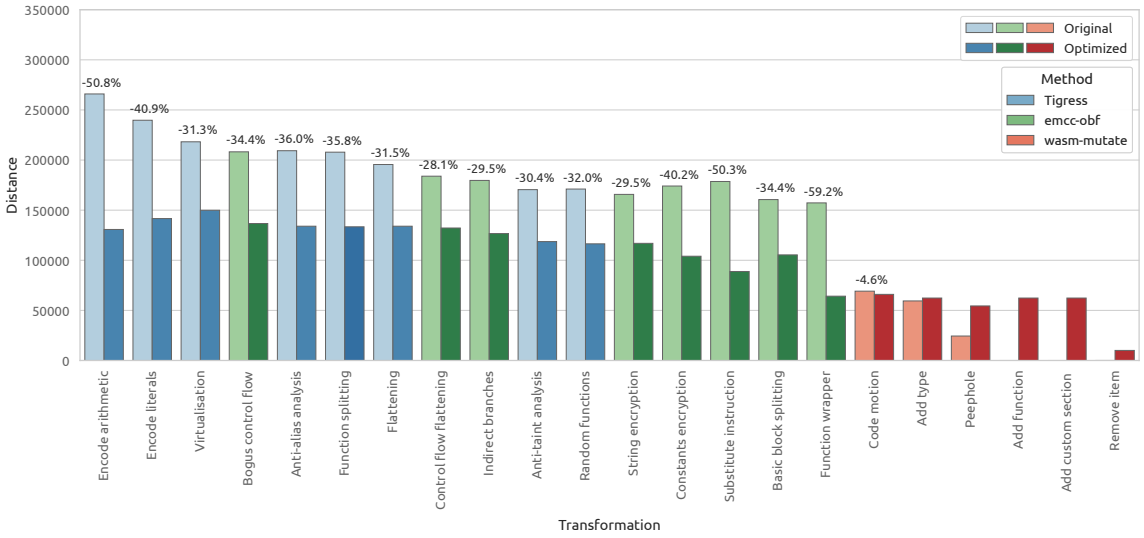


Figure 4.8: Distances for each obfuscation method and transformation before and after de-obfuscation, sorted by the relative decrease in the distance after de-obfuscation in descending order. The percentages show the relative decrease in the distance after de-obfuscation for each transformation. Transformations marked with + indicate an increase in distance after de-obfuscation.

Figure 4.8 shows the distances (Definition 1) for each obfuscation method and transformation before and after automatic de-obfuscation. De-obfuscation was performed using Binaryen's `wasm-opt` optimiser, which averaged an execution time of 1.74 seconds.

Our findings reveal an average reduction in distance of 35.2% after de-obfuscation. The average decrease in distance is 35.9% and 37.3% for Tigress and emcc-obf, respectively, indicating that Tigress is slightly more resilient to de-obfuscation than emcc-obf. Data obfuscation was easier to reverse than control obfuscation, decreasing by 42.3% and 32% on average, respectively. Preventive transformations showed an average decrease around the mean at 34%. We also observed that encryption was more resilient to de-obfuscation than encoding, decreasing 34.8% and 45.8% on average, respectively.

For wasm-mutate, code motion decreased in distance by 4.6%, while the other transformations increased in distance after de-obfuscation. As depicted in Figure 4.9, the stacked transformations lead to progressively larger decreases in distances after de-obfuscation. This indicates that as more transformations are applied, de-obfuscation is more effective.

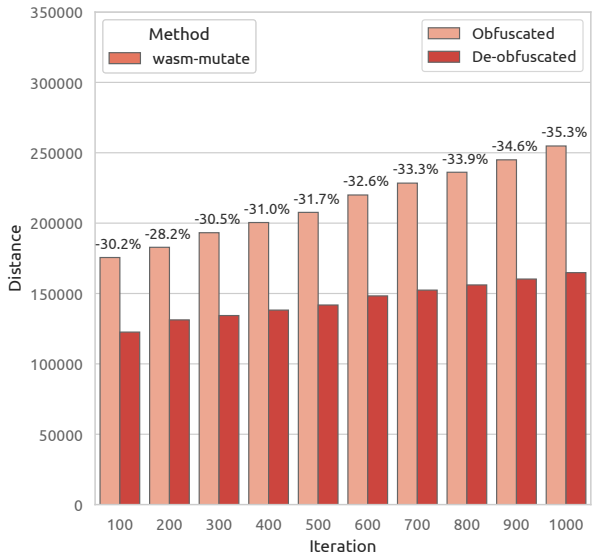


Figure 4.9: Distances for each iteration applied with wasm-mutate before and after de-obfuscation. The percentages show the relative decrease in the distance after de-obfuscation for each transformation.

4.3.2 Detection results after de-obfuscation

Figure 4.10 shows the F_1 scores (Definition 1) for each detection and obfuscation method after de-obfuscation. The results are nuanced, as de-obfuscation can sometimes increase detection accuracy while, in other cases, decreasing it. For MINOS, the accuracy consistently increases after de-obfuscation. The accuracy generally decreases after de-obfuscation for WASim (neural) and WASim (naive).

However, for WASim (RF), accuracy improves significantly, increasing the F_1 score from 0 to 0.80 on average. Another interesting observation is that de-obfuscation tends to be successful for binaries obfuscated with Tigress but not for binaries obfuscated with emcc-obf or wasm-mutate.

Summary De-obfuscation is partially able to reverse the transformations, averaging a 35.2% decrease in distance after de-obfuscation. Binaries obfuscated with Tigress are marginally more resilient than those obfuscated with emcc-obf. Moreover, data obfuscation is easier to reverse than control obfuscation, and encryption is more resilient than encoding. In the case of wasm-mutate, de-obfuscation becomes more effective as more transformations are applied. The impact on detection accuracy is varied; for MINOS, it generally improves and for WASim, it tends to decrease except for the RF classifier, which significantly improves. Interestingly, de-obfuscation can be effective for increasing the detection accuracy, but only for binaries obfuscated with Tigress.

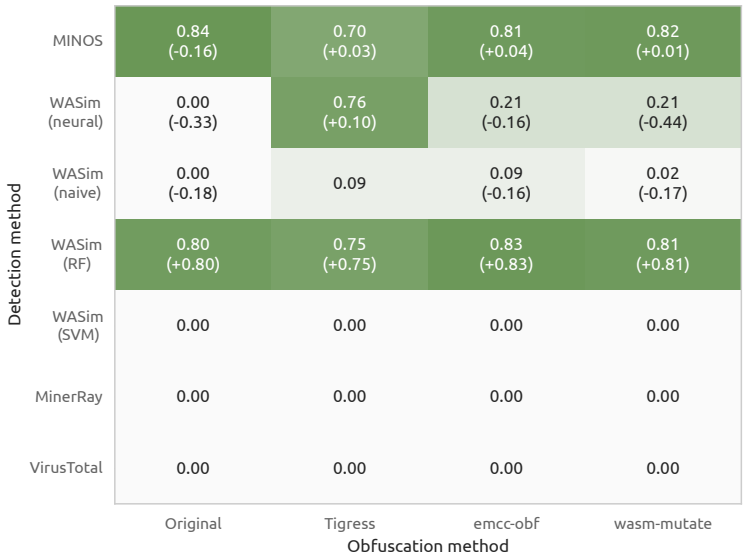


Figure 4.10: F_1 scores for each detection and obfuscation method after de-obfuscation. Dark colours indicate a higher F_1 score, while lighter colours indicate a lower F_1 score. The numbers in the parenthesis signify the difference from Figure 4.5.

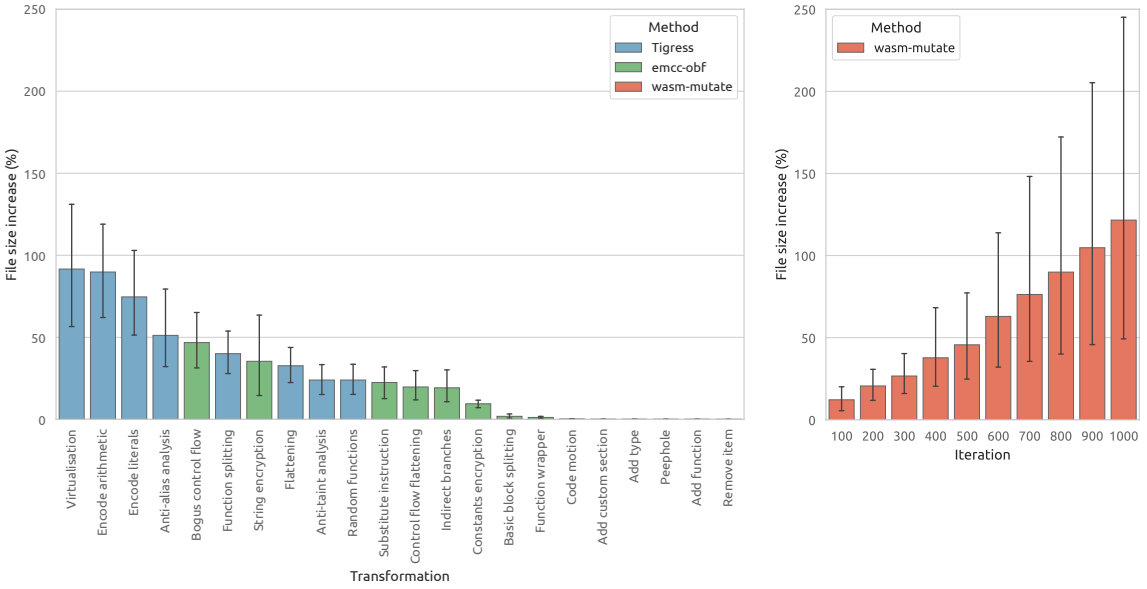
4.4 Overhead

The objective of this section is to address the following research question:

RQ4 – Overhead To what extent do the transformations contribute to overhead in terms of code size and hash rate?

4.4.1 File size overhead

Figure 4.11 shows the relative increase in file size (Definition 6) 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 virtualisation, encode arithmetic, and encode literals, which lead to increases of 91.6%, 89.8%, and 74.6%, respectively. No significant differences were observed between control and data obfuscation in terms of file size overhead.



(**a**) File sizes for each transformation.

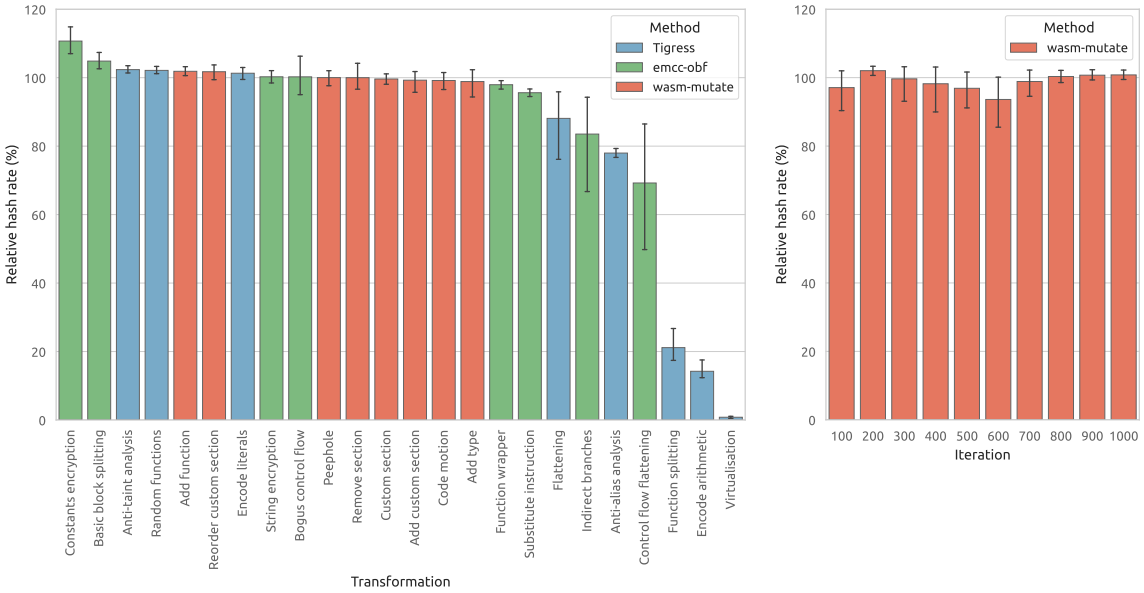
(**b**) File sizes for each iteration.

Figure 4.11: 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.

4.4.2 Hash rate overhead

Figure 4.12 shows the relative hash rate (Definition 8) 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. Two-thirds of the transformations do not significantly impact the hash rate. 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' virtualisation, 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 4.13 shows the relative hash rate (Definition 8) 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 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.



(**a**) Hash rates for each transformation.

(**b**) Hash rates for each iteration.

Figure 4.12: 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.

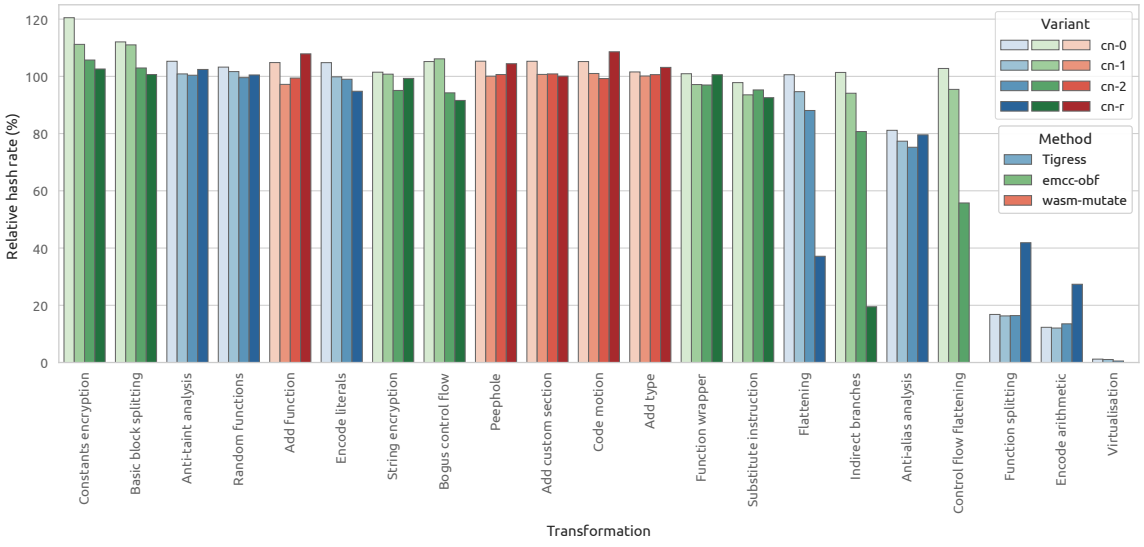


Figure 4.13: Relative hash rates for each obfuscation method, transformation, and CryptoNight variant sorted in descending order.

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' virtualisation 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.

Chapter 5

Discussion

This chapter reflects on the main findings, aligning them with existing literature and offering potential explanations and implications. Moreover, we highlight the limitations of the thesis. The structure remains the same as in the preceding chapter, with each section addressing a research question.

5.1 Effectiveness

Upon analysing the WebAssembly binaries and resulting native code after obfuscation, we find that Tigress is generally the most effective obfuscation method, followed by emcc-obf and wasm-mutate. We hypothesise that this is due to Tigress implementing more advanced transformations, as evidenced by the larger distances, the increased binary size, and the larger native code produced after compilation in the browser. These observations align with the research conducted by Suresh et al. [159], which concluded that Tigress generally applies more advanced transformations than OLLVM, which emcc-obf is based on.

Expanding on this, 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. Importantly, implementing advanced transformations is simpler at higher abstraction levels. For instance, implementing virtualisation would likely be simpler at the source code level, leveraging the powerful abstractions of the programming language, compared to implementing it in the low-level WebAssembly language. On the contrary, lower abstraction levels allow the transformations to target the specific architecture. This is the case for wasm-mutate, enabling direct manipulation of the WebAssembly module, which is not possible at higher abstraction levels such as source code or LLVM bitcode. However, these targeted transformations are not advanced enough to be effective for obfuscation purposes, at least not when applied individually.

The ineffectiveness of individual transformations is primarily due to wasm-mutate being designed as a diversifier rather than an obfuscator. By design, 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 are obfuscators that implement advanced transformations, and thus make significant modifications to the program. However, we find that stacking the wasm-mutate transformations can be an effective approach. This is in line with Cabrera et al.'s study [25], which suggests that a minimum of 120 wasm-mutate iterations are required for successful malware evasion.

Our findings indicate that the most effective transformations are encode arithmetic, encode literals, and virtualisation. On the contrary, Bhansali et al. [17] found anti-alias analysis to be the most effective, followed by virtualisation 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 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 virtualisation 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 underline the crucial influence of the application content on the effectiveness of the transformations. For instance, Figure 4.2 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 4.4, 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.

We would also like to emphasise that the significant error bars, such as those seen in Figure 4.4, are due to the diverse dataset used and resulting different sizes of applications. For instance, games are typically larger than utilities due to the inclusion of external libraries. As longer sequences tend to produce larger distances, this explains why the error bars are large in these plots.

When considering the impact on native code, Tigress consistently produced more native code than emcc-obf and wasm-mutate. However, this increase in native code size is not directly reflected in the WebAssembly binary size. For example, emcc-obf's bogus control flow transformation led to nearly double the WebAssembly binary size increase compared to Tigress' random function transformation, as shown in Figure 4.11. Still, Tigress' random function transformation led to larger native

code, as evident in Figure 4.3. We believe that this discrepancy is due to Liftoff's lazy compilation strategy, which only compiles functions only if they are invoked. 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. In contrast, Tigress generates code that is actually executed and thus compiled to native code, resulting in a smaller WebAssembly binary but larger native code.

An intriguing observation is that TurboFan is more efficient at optimising native code that has been obfuscated using Tigress than with emcc-obf or wasm-mutate. Notice in Figure 4.3 that the relative size of the TurboFan code tends to be smaller than the Liftoff code for Tigress but larger for emcc-obf and wasm-mutate. This is likely due to the fact that the native code compiled from Tigress contains more "junk" code that is easier to optimise away with TurboFan, while the opposite is true for the native code compiled from emcc-obf and wasm-mutate. This hypothesis is supported by the fact that Tigress produces more native code than emcc-obf and wasm-mutate, indeed suggesting that it contains "junk" code to potentially optimise away.

Although TurboFan managed to decrease 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, leaving around a 10% increase in native code size after optimisation. Still, we are uncertain of the specific transformations that were removed since we only have access to the native code's size and not the native code itself, making it impossible to definitively say that obfuscation was not removed. However, considering the significant increase in native code even after optimisation, it is likely that at least some obfuscation remained. These findings contrast studies on LLVM diversification for WebAssembly, where the native code was found to be identical after TurboFan optimisation [7].

5.2 Detectability

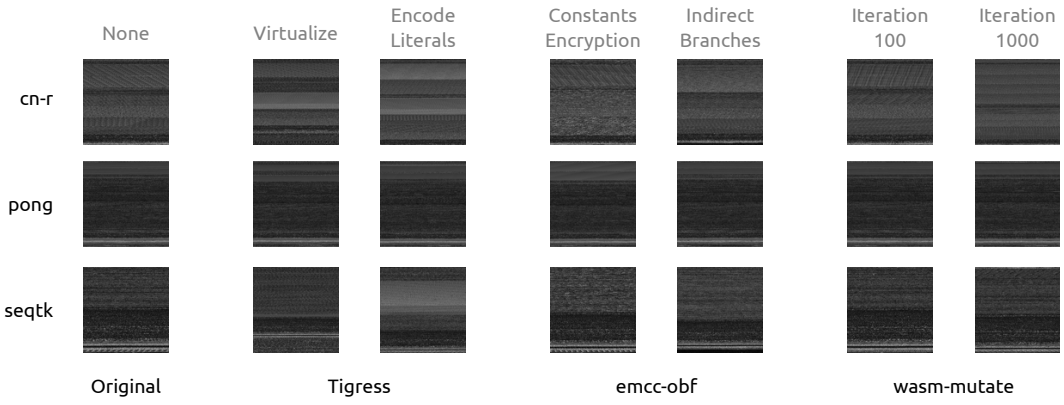


Figure 5.1: Grayscale images of WebAssembly binaries before and after obfuscation. The images serve as input to the CNN used by MINOS.

The experiments indicate that obfuscation can evade state-of-the-art drive-by-mining detectors. This is not surprising and can even be visually seen in Figure 5.1. It shows how the grayscale images of WebAssembly binaries change after obfuscation, which are subsequently used as input to the MINOS classifier. In layman's terms, obfuscation changes the input of the neural network, thereby leading to a different output.

However, it is difficult to precisely identify why and how neural networks are influenced by obfuscation. Nonetheless, we can attempt to reason about why. For instance, MINOS evaluates the entire program by converting it into a grayscale image and then analysing it. As we apply more advanced transformations, the resulting image after obfuscation deviates significantly from the images used in training, leading to the misclassification of the obfuscated program. This hypothesis aligns with the findings of Loose et al. [103], who found that modifying the grayscale image input of MINOS could result in misclassification.

On the other hand, WASim employs a different strategy, examining features such as the WebAssembly binary size. As shown in Table 4.1, the crypto miner probability gradually increases as more transformations are applied up to iteration 500 for WASim. The classifier might learn that crypto miners typically have larger binary sizes than other applications, hence the increase in probability as more transformations are applied since the binary size also increases. Even WASim's authors state that *"a 0.6KB binary module is unlikely to be a Game or crypto miner as those require more functionality than can fit into 0.6KB"* [145]. However, as the binary size continues to increase beyond iteration 500, the crypto miner probability decreases, potentially because the program is deemed too complex and large to be a crypto miner.

As evident in Figure 4.6, we observe that WASim's neural classifier is particularly vulnerable to obfuscation. This is consistent with existing research on adversarial samples [76, 87, 103], demonstrating the sensitivity of neural networks to such adversarial inputs. This also explains why obfuscation is effective for MINOS, since it is also based on neural networks.

The decreased detection rates were often the result of benign applications being mistakenly identified as crypto miners rather than crypto 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. [17], 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 minimise 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, only the data of the program. 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 [17]. However, they used the original MINOS implementation, while we used a reproduction by Cabrera et al. [25], 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.

As for wasm-mutate, it was found to be ineffective in evading detection when applying individual transformations. However, when the transformations were stacked, the accuracy of the detection methods was significantly reduced. We found that code motion and peephole were the most effective transformations. Similarly, Cabrera et al. [25] 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, again highlighting that obfuscation effectiveness is heavily dependent on the specific application being obfuscated.

5.3 Reversibility

When comparing the reversibility of obfuscation methods, emcc-obf appeared to be slightly more reversible than Tigress. This is in contrast to the results of Banescu et al. [12], who found no significant differences between Tigress and OLLVM when it came to symbolic-execution-based de-obfuscation. Despite Tigress exhibiting greater resilience to automatic de-obfuscation based on our distance measures, it was the only obfuscation method that consistently improved in detection accuracy after de-obfuscation. Thus, we do not find a direct correlation between how much obfuscation is reversed and how much the detection accuracy improves.

In the case of wasm-mutate, code motion was the only transformation that decreased in distance after de-obfuscation. For the other transformations, the distances increased after de-obfuscation. We theorise that this occurs because wasm-mutate performs subtle modifications to the WebAssembly binaries, which causes a snowball effect that translates to significant differences after optimisation. When the transformations are stacked, de-obfuscation becomes more effective, likely because there is more "junk" code to eliminate.

Consistent with the findings of Banescu et al. [12], we found data obfuscation to be more reversible than control obfuscation. This is because data obfuscation only affects individual pieces of data, while control obfuscation affects the entire program, making it more difficult to reverse. In addition, compilers often tread carefully around control flow optimisations as they often impact the semantics of the program. Thus, using optimisers, such as Binaryens `wasm-opt`, for de-obfuscation purposes may be ineffective for effectively reversing control obfuscation.

It is also worth noting that there are differences within the reversibility of the various control obfuscation transformations. For instance, bogus control flow introduces dead paths that are never executed, which the compiler is able to identify as useless code and eliminate. On the other hand, reversing control flow flattening back to its original form presents a more substantial challenge, as the original control flow is lost, and the compiler must attempt to reconstruct it.

Moreover, we find data encoding to be more reversible than data encryption. Encoding can be reversed by the compiler because the data is replaced with mathematically equivalent expressions, and compilers are designed for simplifying such expressions. Conversely, encryption transforms the data, making it hard to reverse without knowing the specific algorithm or key used. Even if the decryption code is present in the code and could theoretically be executed by the optimiser, doing so would typically go beyond the scope of what optimisers are designed to do, which is to improve the performance of the code without changing its behaviour.

We found that de-obfuscation could both increase and decrease the accuracy of the detection methods, acting as a double-edged sword. For MINOS, the detection rate improved, likely because after de-obfuscation the resulting grayscale images more closely resembled the images it was trained on, thus increasing the accuracy. The detection accuracies for WASim (neural) and WASim (naive), conversely, decreased, with a significant decline in recall. That is, the number of false negatives increased, indicating that more crypto miners were misclassified as benign programs. This might be due to the de-obfuscated binaries not being complex enough to be classified as crypto miners by WASim.

Interestingly, detection methods that *decreased* in accuracy after obfuscation, like MINOS, *increased* in accuracy after de-obfuscation. We also observed that detection methods that *increased* in accuracy after obfuscation, like WASim, *decreased* in accuracy after de-obfuscation. This implies that the de-obfuscation was effective in removing obfuscation, reverting the effects it had initially caused.

For WASim (RF), we observed a notable increase in the detection F_1 score, from 0 to 0.80 on average, indicating an improvement in the detection of all obfuscation methods. While we can only hypothesise why this is the case, it is plausible that the optimisation removed some noise from the WebAssembly binaries, leaving distinguishable, relevant features in the data that enable effective decision-making at each node of the trees in the forest.

The process of de-obfuscation, on average, took 1.74 seconds to execute. Given the potential improvement in detection accuracy, we believe that this additional time is justifiable. Of course, not all detection methods will benefit from de-obfuscation, as we saw with the WASim (neural) and WASim (naive) detectors. However, the results for MINOS and WASim (RF) show that de-obfuscation can be a useful tool for improving detection accuracy.

5.4 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 [159], wherein the code generated by Tigress was 5% to 78% larger compared to the code generated by OLLVM. It is interesting to see that even for similar transformations, such as Tigress' flatten and emcc-obf control flow flattening, Tigress produces 11.3% larger binaries. We attribute this to Tigress likely using a more complex control flow flattening algorithm.

Cabrera et al. [24] did not measure the size of the WebAssembly binaries that were obfuscated with wasm-mutate. However, we found that stacking transformations can induce considerable overhead, ranging from 12% to 121%, averaging 59%. Still, this is not an issue if the transformations are strategically selected to minimise the overhead or fewer transformations are applied. Such approaches have proven viable in other domains, such as in the diversification of the Rosette language [105].

In terms of performance overhead, Tigress performed the worst, averaging 63.1% of the original hash rate, followed by emcc-obf and wasm-mutate at 95.2% and 99.8%, respectively. These observations echo the findings of Suresh et al. [159], who noted a performance overhead ranging from 4% to 55.4% for code generated by Tigress compared to that generated by OLLVM. Notice in Figure 4.12 that function splitting, encode arithmetic, and virtualisation induce the largest hash rate overheads. Looking at the increase in native code size in Figure 4.3, we find that the same transformations also cause the largest increase in native code size. The correlation here is not surprising, as the more native code that is generated, the more instructions the CPU has to execute, which in turn results in worse performance and a lower hash rate. As such, Tigress applies advanced transformations, which significantly increases how much native code is generated, resulting in a lower hash rate.

Indeed, this reasoning is generally sound, however, an interesting deviation can be seen when examining the native code generated by the stacked wasm-mutate transformations. As Figure 4.4 shows, the native code produced after 1000 stacked wasm-mutate transformations is considerable. However, this is not reflected in Figure 4.13, where the performance overhead is negligible.

We attribute this to hardware optimisations applied to the native code, a process that can diminish the number of instructions executed by the CPU and thus maintain a high hash rate despite a significant increase in native code. Such hardware optimisations can range from pipelining and instruction reordering to branch prediction and cache optimisation, all of which streamline code execution. As for Tigress, we hypothesise that its advanced transformations generate code of such complexity that it prevents the hardware from applying the same degree of optimisation. This results in a larger performance overhead, even when the native code increase is similar to that observed with wasm-mutate.

Another interesting 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 hypothesise that these clean-up procedures inadvertently optimise 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 optimisations. For instance, we found that loop unrolling transformations and code replacements that lead to smaller binaries could increase the hash rate. However, wasm-mutate can also apply transformations that may reduce performance, explaining why the hash rate can also be reduced. This is in line with the findings of Cabrera et al. [24], who found that the performance overhead caused by wasm-mutate fluctuates greatly, ranging from 20% to 147% of the original hash rate.

We have discovered that there are differences between the CryptoNight variants when it comes to the performance overhead. CryptoNightR (cn-r) is architectured to be more resistant to application-specific integrated circuits (ASICs) 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 randomised 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, we find that CryptoNightR has 1.7 times more branch instructions than the other variants, justifying why transformations like flattening, control flow flattening, and indirect branches have a larger impact on it compared to the other variants.

The evolution of the CryptoNight algorithm has led to an increasing complexity to ensure ASIC-resistance. This has a knock-on effect on the performance overhead introduced by obfuscation. As Figure 4.13 depicts, cn-0 is usually the least affected by obfuscation in terms of hash rate, succeeded by cn-1, cn-2, and cn-r. There is a noticeable correlation between the algorithm's complexity and the performance overhead; the more complex algorithms tend to see greater performance overheads. This is because more complex algorithms, with more operations and more intricate control flows, provide more "surface area" for obfuscation to take effect and potentially slow down the program.

5.5 Interpreting the findings

The results derived from this thesis are intricate, and they are influenced by many factors. Although several transformations can evade drive-by mining detection, they often introduce considerable overhead. This raises the question; can obfuscation be used for evading drive-by mining 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, though with 78% of the original hash rate and a 51% file size increase. More desirable results can be achieved by adapting the obfuscation strategy to the specific use case. For instance, 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 merely 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. Intriguingly, in many cases, the performance overhead is negligible, and in some cases, wasm-mutate can even improve the hash rate. Although the file size overhead can be considerable, it can be mitigated by selectively applying transformations that do not substantially increase the file size.

This thesis focuses on crypto mining WebAssembly binaries, but 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 thesis can likely be extended to benefit this domain as well.

5.6 Limitations

The dataset used in this thesis 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. In addition to this, they must be compatible with CIL so they can be parsed and merged into a single source file. Constructing the dataset has been a massive undertaking, and we have spent significant time on it, addressing bugs in Tigress and CIL along the way. Notably, the dataset contains all the prominent CryptoNight variants. This should provide extensive coverage of crypto mining malware, as several studies have found that in-the-wild drive-by mining implementations are all based on the CryptoNight algorithm [25, 84]. 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 [17, 25, 103].

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 2.5.1, 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 thesis.

To our knowledge, there are no de-obfuscation tools for WebAssembly binaries. We, therefore, used a novel approach, using Binaryen's optimiser to de-obfuscate the binaries. However, one should bear in mind that this likely is not the most effective strategy. Developing a de-obfuscation tool specifically for WebAssembly would probably yield even better results. However, we believe this approach is sufficient for this thesis, especially considering the novelty of the field, as it demonstrates that certain transformations are more reversible than others and that the detection accuracy can be improved in certain cases.

We could not extract the native code compiled by the browser, even after conversations with 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 if it is optimised away or not. While we find this useful, we acknowledge that this is a limitation of our research.

Chapter 6

Conclusion

In this thesis, we have conducted an in-depth evaluation of code obfuscation for WebAssembly. The obfuscation methods evaluated operate on multiple abstraction levels, providing the most comprehensive evaluation of WebAssembly obfuscation to date. We have shown how effective obfuscation is at producing dissimilar WebAssembly binaries and how the resulting native code is affected. The results show that obfuscation can, indeed, successfully evade state-of-the-art drive-by mining detectors. However, effectiveness largely depends on the specific obfuscation transformation, detection method, and crypto mining algorithm. In addition to proposing a novel obfuscation method for WebAssembly, we also introduce and evaluate a novel de-obfuscation method based on compiler optimisations. Lastly, we have evaluated the space and time overhead caused by obfuscation and shown how obfuscation can be used in real-world scenarios for drive-by mining evasion with minimal overhead.

These findings are significant for researchers and academics. Notably, we offer insights into which transformations are most effective in evading detection, which can help in developing more advanced detection methods. We also provide an extensive dataset of nearly 50 000 obfuscated WebAssembly binaries for researchers to use, consisting of all the predominant CryptoNight variants, as well as benign applications spanning a diverse set of use cases. Moreover, we have developed novel obfuscation and de-obfuscation methods that can be used in future research.

6.1 Future research

Although this thesis 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 contend with the obfuscation methods explored in this thesis. A promising solution, as seen in this thesis, 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 thesis, 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 thesis. In practice, this could be implemented as optimisation passes for Binaryen, similar to what was done for OLLVM. Moreover, combining several obfuscation transformations, possibly at multiple abstractions levels, merits further investigation.

Finally, there is extensive research on drive-by mining 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 [102], and it has been used for hacking games [73, 11]. 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 in the context of preventing reverse engineering is also an interesting direction for future research.

6.2 Concluding remarks

As we edge towards a future where WebAssembly will take an even more prominent role in web applications, the importance of understanding and countering obfuscation techniques cannot be overstated. This thesis is a step in that direction, providing a stepping stone for academics and researchers to develop advanced detection methods that can withstand obfuscation. We have identified the most effective obfuscation methods and evaluated their feasibility in real-world scenarios by measuring the performance overhead, constituting a significant contribution to the body of literature. Moreover, we provide an extensive dataset of obfuscated WebAssembly binaries, as well as novel obfuscation and de-obfuscation methods for researchers to explore and extend. Despite considerable progress, we are only scratching the surface. We expect the landscape of WebAssembly obfuscation to continue to evolve, and we are confident that our efforts will inspire and instigate further research.

Bibliography

[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] Bertrand Anckaert, Mariusz Jakubowski, Ramarathnam Venkatesan, and Koen De Bosschere. Run-time randomization to mitigate tampering. In *Advances in Information and Computer Security*, pages 153–168. Springer Berlin Heidelberg, 2007.

[4] Alberto Apostolico and Concettina Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[5] Sandhya Armoogum and Asvin Caully. Obfuscation techniques for mobile agent code confidentiality. *Journal of E-Technology*, 1(2):83–94, 2010.

[6] 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.

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

[8] 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.

[9] Nadav Avital. Crypto Me0wing Attacks: Kitty Cashes in on Monero | Imperva. https://www.imperva.com/blog/crypto-me0wing-attacks-kitty-cashes-in-on-monero/?redirect=Incapsula, October 2019. [Accessed 16. May 2023].

[10] Lee Badger, Larry D'Anna, Doug Kilpatrick, Brian Matt, Andrew Reisse, and Tom Van Vleck. Self-protecting mobile agents obfuscation techniques evaluation report. *Network Associates*

Laboratories, Report, pages 01–036, 2002.

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

[12] 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.

[13] Ziv Bar-Joseph, Georg Gerber, David K Gifford, Tommi S Jaakkola, and Itamar Simon. A new approach to analyzing gene expression time series data. In *Proceedings of the sixth annual international conference on Computational biology*, pages 39–48, 2002.

[14] JF Bastien. Going public launch bug. https://github.com/WebAssembly/design/issues/150, June 2015. [Accessed 14. May 2023].

[15] 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.

[16] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, USA:, 1994.

[17] 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.

[18] 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.

[19] Bidragsytere til Wikimedia-prosjektene. ARPANET – Wikipedia. https://no.wikipedia.org/w/index.php?title=ARPANET&oldid=23111496, November 2022. [Accessed 30. May 2023].

[20] 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.

[21] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, feb 2008.

[22] Bramus. Adobe Photoshop in the browser thanks to WASM/Emscripten, Web Components, and Project Fugu. https://www.bram.us/2021/10/27/adobe-photoshop-in-the-browser-thanks-to-emscripten-web-components-and-project-fugu, October 2021. [Accessed 16.

May 2023].

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

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

[25] Javier Cabrera-Arteaga, Martin Monperrus, Tim Toady, and Benoit Baudry. Webassembly diversification for malware evasion. *Computers and Security*, December 2022.

[26] cazala. coin-hive. https://github.com/cazala/coin-hive, March 2018. [Accessed 16. May 2023].

[27] Jien-Tsai Chan and Wuu Yang. Advanced obfuscation techniques for java bytecode. *Journal of Systems and Software*, 71(1-2):1–10, apr 2004.

[28] 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].

[29] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008.

[30] Kevin Cheung. AutoCAD & WebAssembly: Moving a 30 Year Code Base to the Web. *InfoQ*, September 2018.

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

[32] Seongje Cho, Hyeyoung Chang, and Yookun Cho. Implementation of an obfuscation tool for c/c++ source code protection on the XScale architecture. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 406–416. Springer Berlin Heidelberg, 2008.

[33] Chromium. Introduction to Portable Native Client. https://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client, December 2022. [Accessed 2 Dec. 2022].

[34] Melissa Chua and Vivek Balachandran. Effectiveness of android obfuscation on evading anti-malware. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, mar 2018.

[35] 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].

[36] Lin Clark. What makes WebAssembly fast? – Mozilla Hacks - the Web developer blog. https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast, February 2017. [Accessed 13. May 2023].

[37] COBF. COBF - Plexaure. https://www.plexaure.de/cms/index.php?id=cobf, May 2023. [Accessed 24. May 2023].

[38] CodeMorph. CodeMorph Code Obfuscator Features. http://www.sourceformat.com/code-morph.htm, February 2014. [Accessed 24. May 2023].

[39] Frederick B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, oct 1993.

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

[41] 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.

[42] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '98*. ACM Press, 1998.

[43] Contributors to Wikimedia projects. JavaScript - Wikipedia. https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1126827786, November 2001. [Accessed 3 Dec. 2022].

[44] Contributors to Wikimedia projects. ActiveX - Wikipedia. https://en.wikipedia.org/w/index.php?title=ActiveX&oldid=1102963222, August 2022. [Accessed 4 Nov. 2022].

[45] Contributors to Wikimedia projects. Adobe Flash - Wikipedia. https://en.wikipedia.org/w/index.php?title=Adobe_Flash&oldid=1126708043, December 2022. [Accessed 2 Dec. 2022].

[46] Contributors to Wikimedia projects. Java (programming language) - Wikipedia. https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1126888277, December 2022. [Accessed 2 Dec. 2022].

[47] Contributors to Wikimedia projects. Dynamic time warping - Wikipedia. https://en.wikipedia.org/w/index.php?title=Dynamic_time_warping&oldid=1150247571, April 2023. [Accessed 26. May 2023].

[48] Contributors to Wikimedia projects. Euclidean distance - Wikipedia. https://en.wikipedia.org/w/index.php?title=Euclidean_distance&oldid=1153733276, May 2023. [Accessed 26. May 2023].

[49] Contributors to Wikimedia projects. History of the Internet - Wikipedia. https://en.wikipedia.org/w/index.php?title=History_of_the_Internet&oldid=1153054494, May 2023. [Accessed 13. May 2023].

[50] Contributors to Wikimedia projects. Jeff Atwood - Wikipedia. https://en.wikipedia.org/w/index.php?title=Jeff_Atwood&oldid=1151929413, April 2023. [Accessed 26. May 2023].

[51] Contributors to Wikimedia projects. Packet switching - Wikipedia. https://en.wikipedia.org/w/index.php?title=Packet_switching&oldid=1159621019, June 2023. [Accessed 13. Jun. 2023].

[52] Contributors to Wikimedia projects. Precision and recall - Wikipedia. https://en.wikipedia.org/w/index.php?title=Precision_and_recall&oldid=1157370599, May 2023. [Accessed 29. May 2023].

[53] CShroud. SourceForge. https://sourceforge.net/projects/cshroud, January 2013. [Accessed 24. May 2023].

[54] d35ha. xObf. https://github.com/d35ha/xObf, May 2023. [Accessed 24. May 2023].

[55] 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.

[56] WebAssembly docs. Security - WebAssembly. https://webassembly.org/docs/security, May 2023. [Accessed 14. Jun. 2023].

[57] S. Drape, O. De Moor, and G. Sittampalam. Transforming the .net intermediate language using path logic programming. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'02)*, pages 133–144, 2002. Cited By :19.

[58] Stephen Drape. Generalising the array split obfuscation. *Information Sciences*, 177(1):202–219, jan 2007.

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

[60] 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].

[61] Electron.js. Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron. https://www.electronjs.org, May 2023. [Accessed 16. May 2023].

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

[63] EOSIO. EOS Virtual Machine: A High-Performance Blockchain WebAssembly Interpreter – EOSIO. https://eos.io/news/eos-virtual-machine-a-high-performance-blockchain-webassembly-interpreter, January 2021. [Accessed 9. Nov. 2022].

[64] EXECryptor. EXECryptor. https://execryptor.en.softonic.com, May 2023. [Accessed 24. May 2023].

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

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

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

[68] Flutter. Support for WebAssembly (Wasm). https://docs.flutter.dev/platform-integration/web/wasm, May 2023. [Accessed 16. May 2023].

[69] S. Forrest, A. Somayaji, and D.H. Ackley. Building diverse computer systems. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*. IEEE Comput. Soc. Press, 2022.

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

[71] Kazuhide Fukushima, Shinsaku Kiyomoto, and Toshiaki Tanaka. An obfuscation scheme using affine transformation and its implementation. *IPSJ Digital Courier*, 2:498–512, 2006.

[72] 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.

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

[74] Dan Goddin. Now even YouTube serves ads with CPU-draining cryptocurrency miners. https://arstechnica.com/information-technology/2018/01/now-even-youtube-serves-ads-with-cpu-draining-cryptocurrency-miners, May 2023. [Accessed 16. May 2023].

[75] Nishu Goel. JavaScript | 2021 | The Web Almanac by HTTP Archive. *2021 Web Almanac*, 3(2), December 2022.

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

[77] Google. Native Client - Chrome Developers. https://developer.chrome.com/docs/native-client, January 1980. [Accessed 4 Nov. 2022].

[78] Google. WebAssembly compilation pipeline · V8. https://v8.dev/docs/wasm-compilation-

pipeline, May 2023. [Accessed 16. May 2023].

[79] Samuel Gross. Samuel Groß and Amy Burnett. 2022. Attacking JavaScriptEngines in 2022. https://saelo.github.io/presentations/offensivecon_22_attacking_javascript_engines.pdf, 2022. [Accessed 13. May 2023].

[80] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

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

[82] Muhammad Hataba, Reem Elkhouly, and Ahmed El-Mahdy. Diversified remote code execution using dynamic obfuscation of conditional branches. In *2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*. IEEE, jun 2015.

[83] Kotlin Help. Get started with Kotlin/Wasm in IntelliJ IDEA | Kotlin. https://kotlinlang.org/docs/wasm-get-started.html, May 2023. [Accessed 16. May 2023].

[84] 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.

[85] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72–93, dec 2018.

[86] T.W. Hou, H.Y. Chen, and M.H. Tsai. Three control flow obfuscation methods for java software. *IEE Proceedings - Software*, 153(2):80, 2006.

[87] Sandy Huang, Nicolas Papernot, Ian Goodfellow, Yan Duan, and Pieter Abbeel. Adversarial attacks on neural network policies. *arXiv preprint arXiv:1702.02284*, February 2017.

[88] Solomon Hykes. Solomon Hykes / @shykes@hachyderm.io on Twitter. https://twitter.com/solomonstre/status/1111004913222324225, March 2019. [Accessed 16. May 2023].

[89] Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei Saw, and Ramarathnam Venkatesan. The superdiversifier: Peephole individualization for software protection. In *Advances in Information and Computer Security*, pages 100–120. Springer Berlin Heidelberg, 2008.

[90] 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.

[91] 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.

[92] Conor Kelton, Aruna Balasubramanian, Ramya Raghavendra, and Mudhakar Srivatsa. Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy. In *Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web*. Internet Society, 2020.

[93] 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.

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

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

[96] Aniket Kulkarni and Ravindra Metta. A new code obfuscation scheme for software protection. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. IEEE, apr 2014.

[97] Shannon Liao. Showtime websites secretly mined user CPU for cryptocurrency. *Verge*, September 2017.

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

[99] 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.

[100] C.-T Lin, N.-J Wang, Han Xiao, and Claudia Eckert. Feature selection and extraction for malware classification. *Journal of Information Science and Engineering*, 31:965–992, 05 2015.

[101] 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.

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

[103] Nils Loose, Felix Mächtle, Claudius Pott, Volodymyr Bezsmertnyi, and Thomas Eisenbarth. Madvex: Instrumentation-based adversarial attacks on machine learning malware detection. *Detection of Intrusions and Malware & Vulnerability Assessment*, May 2023.

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

[105] 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.

[106] Marcelo De A Maia, Victor Sobreira, Klérisson R Paixão, SA Amo, and Ilmério R Silva. Using a sequence alignment algorithm to identify specific and common code from execution traces. In *Proceedings of the 4th International Workshop on Program Comprehension through Dynamic Analysis (PCODA*, pages 6–10, 2008.

[107] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers Security*, 51:16–31, jun 2015.

[108] A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Conferences in Research and Practice in Information Technology Series*, volume 48, pages 187–196, 2006. Cited By :30.

[109] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. WebAssembly modules as lightweight containers for liquid IoT applications. In *International Conference on Web Engineering*, pages 328–336. Springer, 2021.

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

[111] Microsoft. Blazor | Build client web apps with C# | .NET. https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor, May 2023. [Accessed 16. May 2023].

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

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

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

[115] 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.

[116] 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.

[117] Mozilla. asm.js - Game development | MDN. https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js?source=post_page, November 2022. [Accessed 4 Nov. 2022].

[118] Mozilla. Using the WebAssembly JavaScript API - WebAssembly | MDN. https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API, November 2022. [Accessed 29 Nov. 2022].

[119] Abdullah Mueen and Eamonn Keogh. Extracting optimal performance from dynamic time warping. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 2129–2130, 2016.

[120] 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.

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

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

[123] NEAR. What is a Smart Contract? | NEAR Documentation. https://docs.near.org/develop/contracts/whatisacontract, November 2022. [Accessed 11 Nov. 2022].

[124] 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.

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

[126] Nikjoo. x86-Code-Virtualizer. https://github.com/NIKJOO/x86-Code-Virtualizer, May 2023. [Accessed 24. May 2023].

[127] Node.js. Node.js. https://nodejs.org/en, November 2022. [Accessed 27 Nov. 2022].

[128] Matija Novak, Mike Joy, and Dragutin Kermek. Source-code similarity detection and detection tools used in academia. *ACM Transactions on Computing Education*, 19(3):1–37, may 2019.

[129] objobf. objobf-0.5.0.tar.bz2 ≈ Packet Storm. https://packetstormsecurity.com/files/

31524/objobf-0.5.0.tar.bz2.html, May 2023. [Accessed 24. May 2023].

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

[131] Ahmet Okutan. Use of source code similarity metrics in software defect prediction. *arXiv preprint arXiv:1808.10033*, 2018.

[132] Oreans. Oreans Technologies : Software Security Defined. https://www.oreans.com/Themida.php, May 2023. [Accessed 24. May 2023].

[133] 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.

[134] Helen Partz. Cybersecurity Firm Detects Cryptojacking Malware on Make-A-Wish Foundation Website. *Cointelegraph*, November 2018.

[135] Mathias Payer. Embracing the new threat: Towards automatically self-diversifying malware. In *The Symposium on Security for Asia Network*, pages 1–5, 2014.

[136] politoinc. Automated Obfuscation of Windows Malware and Exploits Using O-LLVM. https://www.politoinc.com/post/2020/03/02/automated-obfuscation-of-windows-malwareexploits-using-o-llvm, March 2020. [Accessed 25. May 2023].

[137] 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.

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

[139] Hardware-Based Protections. A survey of anti-tamper technologies. *CrossTalk: The J. Defense Softw. Engin*, 2004.

[140] Enigma Protector. Software Protection, Software Licensing, Software Virtualization. https://enigmaprotector.com, April 2023. [Accessed 24. May 2023].

[141] Pythondev. Compile Python to WebAssembly (WASM) — Unofficial Python Development (Victor's notes) documentation. https://pythondev.readthedocs.io/wasm.html, April 2023. [Accessed 16. May 2023].

[142] Chotirat Ann Ratanamahatana and Eamonn Keogh. Everything you know about dynamic time warping is wrong. In *Third workshop on mining temporal and sequential data*, volume 32. Citeseer, 2004.

[143] 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.

[144] 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.

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

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

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

[148] Dănuţ Rusu. Protection methods of java bytecode. In *Proceedings of the Networking in Education and Research International Conference*, pages 214–220, 2003.

[149] Hiroaki Sakoe and Seibi Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

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

[151] 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.

[152] Federico Scrinzi. Behavioral analysis of obfuscated code. Master's thesis, University of Twente, 2015.

[153] Marija Selakovic and Michael Pradel. Performance issues and optimizations in javascript: An empirical study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 61–72, New York, NY, USA, 2016. Association for Computing Machinery.

[154] Liang Shan and Sabu Emmanuel. Mobile agent protection with self-modifying code. *Journal of Signal Processing Systems*, 65(1):105–116, nov 2010.

[155] Praveen Sivadasan, P SojanLal, and Naveen Sivadasan. Jdatatrans for array obfuscation in java source codes to defeat reverse engineering from decompiled codes. In *Proceedings of the 2nd Bangalore Annual Compute Conference*, pages 1–4, 2009.

[156] Quentin Stiévenart and Coen De Roover. Compositional information flow analysis for webassembly programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 13–24. IEEE, 2020. WASSAIL.

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

[158] Stunnix. C/C++ Obfuscator. http://stunnix.com/prod/cxxo, May 2023. [Accessed 24. May 2023].

[159] 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.

[160] 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.

[161] Romain Tavenard. An introduction to dynamic time warping. https://rtavenar.github.io/blog/dtw.html, 2021.

[162] Oreans Technologies. Oreans Technologies : Software Security Defined. https://www.oreans.com/CodeVirtualizer.php, May 2023. [Accessed 24. May 2023].

[163] W3 Techs. Usage Statistics of JavaScript as Client-side Programming Language on Websites, December 2022. https://w3techs.com/technologies/details/cp-javascript, November 2022. [Accessed 4 Nov. 2022].

[164] 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].

[165] W. Thompson, A. Yasinsac, and T. McDonald. Semantic encryption transformation scheme. In *17th ISCA International Conference on Parallel and Distributed Computing Systems 2004, PDCS 2004*, pages 516–521, 2004. Cited By :9.

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

[167] Marco Trivellato. WebAssembly is here! | Unity Blog. https://blog.unity.com/technology/webassembly-is-here, August 2018. [Accessed 16. May 2023].

[168] Can I Use? WebAssembly | Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/wasm, November 2022. [Accessed 12 Nov. 2022].

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

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

[171] VMProtect. VMProtect | F-Secure Labs. https://www.f-secure.com/v-descs/

vmprotect.shtml, May 2023. [Accessed 24. May 2023].

[172] W3. Same Origin Policy - Web Security. https://www.w3.org/Security/wiki/Same_Origin_Policy, November 2022. [Accessed 26 Nov. 2022].

[173] 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].

[174] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Technical Report CS-2000-12, University of Virginia, 12 2000, 2000.

[175] 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.

[176] 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.

[177] WASI. WASI |. https://wasi.dev, June 2022. [Accessed 25 Nov. 2022].

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

[179] Wavm. WAVM. https://github.com/WAVM/WAVM, May 2023. [Accessed 16. May 2023].

[180] WebAssembly. Security - WebAssembly. https://webassembly.org/docs/security/#users, November 2022. [Accessed 3 Nov. 2022].

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

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

[183] 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.

Appendices

Appendix A

Repository: wasm-obf

This appendix contains the source code and experimental data derived from this thesis. The source code is publicly available on GitHub,[1] and the data can be found under the releases page.[2]

README

This repository contains the source code and experimental data derived from research on WebAssembly obfuscation. It has been developed as a part of my Master's thesis in Computer Science at the Norwegian University of Science and Technology (NTNU). The experimental data, containing close to 50,000 WebAssembly binaries, can be found under the releases.

Structure

- Analysis contains the data and code to create the plots used in the thesis.
- Dataset contains the source code and build files for the applications in the dataset.
- Detection contains the source code of the cryptomining detection methods.
- Metrics contains code for measuring the file size, hash rate, and similarity between WebAssembly binaries.
- Miner contains code for the web-based cryptominer.
- Mongodb contains code relating to the mongodb database.
- Obfuscation contains code for obfuscating the WebAssembly binaries.
- Optimization contains code for optimizing the WebAssembly binaries.
- Verify hashes contains code for verifying the hashes of the cryptomining WebAssembly binaries.

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

Requirements

- Python 3
- Docker

Setup

Some of the docker containers require specific networks to be setup. Specifically, a database, miner, and WASim network will need to be created:

```
docker create network db_network
docker create network mn_network
docker create network wasim_network
```

Usage

Starting the database

Navigate to the mongodb folder and start the docker container:

```
docker compose run mongodb
```

Building the dataset

Navigate to the dataset folder and build the dataset:

```
docker compose run build-dataset
```

This will build the applications in the dataset folder using Emscripten and move the WebAssembly binaries, as well as the accompanying JavaScript glue code and HTML file to the binaries folder.

Obfuscating the WebAssembly binaries

Navigate to the obfuscation folder and run:

```
docker compose run <method>
```

where `<method>` is either `tigress`, `llvm`, or `wasm-mutate`.

Running cryptomining detection

Navigate to the detection folder and run

```
docker compose run <method>
```

where `<method>` is either `minos`, `miner-ray`, `virustotal`, or `wasim`.

Measuring file size and distance

Navigate to `metrics` and run:

```
docker compose run file-size
docker compose run dtw
```

Measure the hash rate

Navigate to `miner` and start it:

```
docker compose up
```

Then, navigate back to `metrics` and run:

```
docker compose run hash-rate
```

Verifying the hashes

Navigate to `verify-hashes` and run:

```
docker compose run verify-hashes
```

Extracting V8 bytecode

Navigate to `optimization` and run:

```
docker compose run v8-stats
```

Optimizing the WebAssembly binaries

Navigate to `optimization` and run:

```
docker compose run opt
```

The database will then need to be reset before re-running the experiment with the optimized WebAssembly binaries.

Appendix B

Repository: emcc-obf

This appendix contains the source code and build scripts for the novel obfuscation method developed in this thesis, emcc-obf. These resources are publicly available on GitHub.[1]

README

This repository contains a modified version of the Emscripten compiler that includes an LLVM-based obfuscator. Specifically, it uses the Hikari obfuscator which is based on the obfuscator-llvm project.

Disclaimer: Some of the documentation is translated from chinese to the best of my ability, sorry!

Table of Contents

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

Usage

The flags operate at the LLVM-level and have to be passed to Emscripten through the `-mllvm` flag. For instance, if you want to add bogus control flow and set the probability to 100% for each basic block, you would have to do:

```
emcc -mllvm -enable-bcfobf -mllvm -bcf_prob 100 <file>.c
```

To only obfuscate certain functions, see Function Annotations.

Note: You may need to turn off optimization so that the obfuscation is not optimized away by the compiler.

Building

Building with Docker

```
docker build -t emcc-obf .
docker run -it emcc-obf
```

Building from source

Emscripten does not require compilation as it uses Python. However, the LLVM (which provides `Clang` and `wasm-ld`) and Binaryen components need to be compiled. Once compiled, you can simply modify the `.emscripten` file to specify the correct paths for these tools using the `LLVM_ROOT` and `BINARYEN_ROOT` variables. These variables may already be correct depending on the output of `emcc --generate-config`. Also, for convenience the Emscripten folder should be added to your path.

Note that `ninja install` installs the compiled binaries in the appropriate directories (usually `/usr/local/bin`), which may conflict with existing installations. If you've already installed LLVM and Binaryen, omit the `ninja install` command and edit the `.emscripten` file accordingly.

Dependencies

Building locally requires the following dependencies:

- gcc
- cmake
- ninja
- nodejs
- python3

Building LLVM

```
git clone --recursive -b llvm-16.0.0rel https://github.com/61bcdefg/Hikari-LLVM15
.git hikari
cd hikari
git submodule update --remote --recursive
mkdir build && cd build
cmake -G "Ninja" -DCMAKE_BUILD_TYPE=MinSizeRel -DLLVM_APPEND_VC_REV=on
-DLLVM_ENABLE_PROJECTS='lld;clang' -DLLVM_TARGETS_TO_BUILD="host;WebAssembly"
-DLLVM_INCLUDE_EXAMPLES=OFF -DLLVM_INCLUDE_TESTS=OFF -DENABLE_LLVM_SHARED=1
../llvm
ninja && ninja install
```

Building Binaryen

```
git clone https://github.com/WebAssembly/binaryen.git binaryen
cd binaryen
git checkout ecbebfbee12f2f25af648119604915fc37427f6f
git submodule init
git submodule update
mkdir build && cd build
cmake -G "Ninja" ..
ninja && ninja install
```

Configure emscripten

```
git clone https://github.com/emscripten-core/emscripten.git emscripten
git checkout fab93a2bff6273c882b0c7fb7b54eccc37276e03
emcc --generate-config
npm i
```

License

See Hikari/License.

Appendix C

SoK: Analysis Techniques for WebAssembly

This appendix contains the preliminary version of the paper *SoK: Analysis Techniques for WebAssembly*. The paper originates from the literature review conducted last semester as part of the *TDT4501 – Specialization Project* course.

SoK: Analysis Techniques for WebAssembly

Håkon Harnes
NTNU

Donn Morrison
NTNU

Abstract

WebAssembly is a low-level bytecode language that allows high-level languages like C, C++, and Rust to be executed in the browser at near-native performance. In recent years, WebAssembly has gained widespread adoption is now natively supported by all modern browsers. However, vulnerabilities in memory-unsafe languages, like C and C++, can translate into vulnerabilities in WebAssembly binaries. Unfortunately, most WebAssembly binaries are compiled from such memory-unsafe languages, and these vulnerabilities have been shown to be practical in real-world scenarios. WebAssembly smart contracts have also been found to be vulnerable, causing significant financial loss. Additionally, WebAssembly has been used for malicious purposes like cryptojacking. To address these issues, several analysis techniques for WebAssembly binaries have been proposed. In this paper, we conduct a comprehensive literature review of these techniques and categorize them based on their analysis strategy and objectives. Furthermore, we compare and evaluate the techniques using quantitative data, highlighting their strengths and weaknesses. In addition, one of the main contributions of this paper is the identification of future research directions based on the thorough literature review conducted.

1 Introduction

The Internet has come a long way since its inception and one of the key technologies that have enabled its growth and evolution is JavaScript. JavaScript, which was developed in the mid-1990s, is a programming language that is widely used to create interactive and dynamic websites. It was initially designed to enable basic interactivity on web pages, such as form validation and image slideshows. However, it has evolved into a versatile language that is used to build complex web applications. Today, JavaScript is one of the most popular programming languages in the world, currently being used by 98% of all websites [17].

Despite its popularity and versatility, JavaScript has some inherent limitations that have become apparent as web applications have grown more complex and resource-demanding. Specifically, JavaScript is a high-level, interpreted, dynamically typed language, which fundamentally limits its performance. Consequently, it is not suited for developing resource-demanding web applications. To address the shortcomings of JavaScript, several technologies, like ActiveX [30], NaCl [1], and asm.js [8], have been developed. However, these technologies have faced compatibility issues, security vulnerabilities, and performance limitations.

WebAssembly was developed by a consortium including Mozilla, Microsoft, Apple, and Google, as a solution to the limitations of existing technologies. WebAssembly is designed as a safe, fast, and portable compilation target for high-level languages like C, C++ , and Rust, allowing them to be executed with near-native performance in the browser. It has gained widespread adoption and is currently supported by 96% of all browser instances [22]. Moreover, WebAssembly is also being extended to desktop applications [59], mobile devices [64], cloud computing [12], blockchain Virtual Machines (VMs) [3, 4, 23], IoT [53, 56], and embedded devices [70].

However, WebAssembly is not without its own set of challenges. Vulnerabilities in memory-unsafe languages, like C and C++, can translate into vulnerabilities in WebAssembly binaries [48]. Unfortunately, two-thirds of WebAssembly binaries are compiled from memory-unsafe languages [41], and these attacks have been found to be practical in real-world scenarios [48]. Additionally, vulnerabilities have been uncovered in WebAssembly smart contracts [42, 63], consequently causing significant financial loss. WebAssembly has also been used for malicious purposes such as cryptojacking [60]. To mitigate these issues, several analysis techniques for WebAssembly binaries have been proposed.

In this paper, we conduct an in-depth literature review of analysis techniques for WebAssembly binaries. To this end, we classify the analysis techniques based on their analysis strategy and objectives. Specifically, we find the analysis techniques can be classified into three categories: detecting malicious WebAssembly binaries (§4.1), detecting vulnerabilities

in WebAssembly binaries (§4.2), and detecting vulnerabilities in WebAssembly smart contracts (§4.3). Moreover, we compare and evaluate the techniques using quantitative data, highlighting their strengths and weaknesses. In addition, one of the main contributions of this paper is the identification of future research directions based on the thorough literature review conducted. In summary, this paper contributes the following:

- A comprehensive analysis of current analysis techniques for WebAssembly binaries, using quantitative data to evaluate their strengths and weaknesses.
- A taxonomical classification of current analysis techniques for WebAssembly binaries.
- Key findings and limitations of current analysis techniques for WebAssembly binaries, including the trade-offs between accuracy and overhead of static and dynamic analysis methods.
- Identification of gaps in the literature and suggestions for future research directions.

2 Background

The background section of this paper provides a detailed overview of WebAssembly. The limitations of JavaScript and prior attempts at incorporating low-level code on the web are first discussed. Then, an in-depth description of WebAssembly's security mechanisms, vulnerabilities, and use cases are presented.

2.1 History

JavaScript Initially, the internet was primarily used by researchers, scientists, and other academics for the purpose of sharing information and collaborating on projects. At this time, websites were mostly composed of static text and images, lacking dynamic or interactive components. The arrival of web browsers such as Netscape Navigator and Internet Explorer in the late 1990s made the internet accessible to the general public and sparked the development of technology to enhance website user experience with dynamic and interactive elements. JavaScript, created by Netscape in 1995 [29], became one of these technologies, enabling web developers to create engaging content. Today, JavaScript is a widely used programming language supported by all major web browsers and used on 98% of websites [17].

Despite its popularity and versatility, JavaScript has some inherent limitations that impact its performance. As a high-level language, JavaScript abstracts away many of the details of the underlying hardware, making it easier to write and understand. However, this also means that the JavaScript engine has to do more work to translate the code into machine-readable instructions. Additionally, because JavaScript is an interpreted language, it must be parsed and interpreted every time it is executed, which can add overhead and decrease performance. Lastly, JavaScript is dynamically typed, meaning the type of a variable is determined at runtime. This can make it difficult for the JavaScript engine to optimize the code, resulting in reduced performance. These limitations can hinder the performance of JavaScript in resource-demanding or complex applications. There is, therefore, a need for high-performance, low-level code on the web.

ActiveX ActiveX [30] is a deprecated framework that was introduced by Microsoft in 1996. It allowed developers to embed signed x86 binaries through ActiveX controls. These controls were built using the Component Object Model (COM) specification, which was intended to make the controls platform-independent. However, ActiveX controls contain compiled x86 machine code and calls to the standard Win32 API, restricting them to x86-based Windows machines. Additionally, they were not run in a sandboxed environment, consequently allowing them to access and modify system resources. In terms of security, ActiveX relied entirely on code signing, and thus did not achieve safety through technical construction, but through a trust model.

NaCl Native Client (NaCl) [1] is a system introduced by Google in 2011 that allows for the execution of machine code on the web. The sandboxing model implemented by NaCl enables the coexistence of NaCl code with sensitive data within the same process. However, NaCl is specifically designed for the x86 architecture, limiting its portability. To address this limitation, Google introduced Portable Native Client (pNaCl) [13] in 2013. pNaCl builds upon NaCl's sandboxing techniques and uses a stable subset of LLVM bitcode as an interchangeable format, allowing for the portability of applications across different architectures. However, pNaCl does not significantly improve compactness and still exposes details specific to compilers and architectures, such as the layout of the call stack. NaCl and pNaCl are exclusively available in Google Chrome, thus limiting the portability of applications that use them.

Asm.js Asm.js [8], which was introduced by Mozilla in 2013, is a strict subset of JavaScript that can be used as an efficient compilation target for high-level languages like C and C++. Through the Emscripten toolchain [14] , these languages can be compiled to asm.js and subsequently executed on modern JavaScript execution engines, benefitting from sophisticated Just In Time (JIT) compilers. This allows for near-native performance. However, the strict subset nature of asm.js means that extending the language with new features, such as `int64`, requires first extending JavaScript and then ensuring compatibility with the asm.js subset. Even then, it can be challenging to implement these features efficiently.

Java and Flash It is also worth noting that Java and Flash were among the first technologies to be used on the web, being released in 1995 and 1996, respectively [31, 32]. They offered

managed runtime plugins; however, neither was capable of supporting high-performance, low-level code. Moreover, their usage has declined due to security vulnerabilities and performance issues.

2.2 WebAssembly

Overview WebAssembly is a technology that aims to address performance, compatibility, and security issues that have plagued previous approaches. It was developed by a consortium of tech companies, including Mozilla, Microsoft, Apple, and Google, and was released in 2017 [38]. Wasm has since gained widespread adoption and is currently supported by 96% of all browser instances [22]. Additionally, it is an official World Wide Web Consortium (W3C) standard [2], and is natively supported on the web.

WebAssembly is a low-level bytecode language, that is executed on a stack-based Virtual Machine (VM). More specifically, instructions pop their inputs from and push their results to an implicit evaluation stack. There are no registers in this system, but individual values can be stored in global variables that are visible to the entire module, or in local variables that are only visible to the current function. The evaluation stack, global variables, and local variables are all managed by the VM.

Host Environment WebAssembly modules are executed in a host environment, which provides the necessary functionality for the module to perform actions such as I/O or network access. In a browser, the host environment is provided by the JavaScript engine, such as V8 or SpiderMonkey. WebAssembly exports can be wrapped in JavaScript functions using the WebAssembly JavaScript API [18], allowing them to be called from JavaScript code. Similarly, WebAssembly code can import and call JavaScript functions. Other host environments for WebAssembly include server-side environments like Node.js [62] and stand-alone VMs, which provide their own APIs for WebAssembly modules to use. For instance, modules running on a stand-alone VM may interact with the local file system through the WebAssembly System Interface (WASI) [20].

Module WebAssembly programs are organized into modules, which are the basic unit of deployment, loading, and compilation. Each module contains definitions for types, functions, tables, memories, and globals. In addition, a module can declare imports and exports, as well as provide initialization through data and element segments or a start function.

Compilation High-level languages like C, C++, and Rust can be compiled to Wasm, since it is designed as a compilation target. Toolchains like Emscripten [14] or wasm-pack [21] can be used to compile these languages to Wasm. The resulting binary is in the `wasm` binary format, but can also be represented in a human-readable text format called `wat`. A module corresponds to one file. The WebAssembly Binary Toolkit (WABT) [81] provides tools for converting between `wasm` and `wat` representations, as well as for decompilation and validation of Wasm binaries. Figure 1 shows an example of code in C, its corresponding representation in `wat`, and its compiled representation in the `wasm` format.

Use Cases WebAssembly has been adopted for various applications on the web due to its near-native execution performance, such as data compression, game engines, and natural language processing. However, the usage of WebAssembly is not only limited to the web. It is being extended to desktop applications [59], mobile devices [64], cloud computing [12], IoT [53,56], and embedded devices [70]. In recent years, WebAssembly has also been adopted by blockchain platforms, such as EOSIO [3] and NEAR [23], as their smart contract runtime. Even Ethereum has placed WebAssembly on the Ethereum 2.0 roadmap as a replacement to the Ethereum Virtual Machine (EVM) [4].

2.2.1 Security

Environment WebAssembly modules execute within a sandboxed environment separated from the host runtime using fault isolation techniques. This implies that modules execute independently, and can't escape the sandbox without going through appropriate APIs. For instance, WebAssembly modules in a web browser have no direct access to the Document Object Model (DOM), but must use JavaScript APIs to interact with it. Additionally, each module is subject to the security policies of its embedding, such as the Same Origin Policy (SOP) [15] enforced by web browsers, which restricts the flow of information between web pages from different origins. In the case of standalone WebAssembly runtimes with operating system support, the module must use proposed APIs to access system resources such as files.

Memory Unlike native binaries, which have access to all of the memory allocated to the process, WebAssembly modules are restricted to a contiguous region of memory known as linear memory. This memory is untyped and byte-addressable, and its size is determined by the data present in the binary. The size of linear memory is always a multiple of a WebAssembly page, which is 64KiB. When a WebAssembly module is instantiated, it uses the appropriate API call to create the necessary memory objects for its execution. The JavaScript engine or system runtime then creates a managed buffer, such as an `ArrayBuffer`, to store the linear memory. This means that the WebAssembly module accesses the physical memory indirectly through the managed buffer, which ensures that it can only read and write data within a limited area of the memory.

Control Flow Integrity Unlike other assembly-like languages, WebAssembly features structured control flow. Instructions in a function are organized into well-nested blocks. Branches can only jump to the end of surrounding blocks, and

```
int square(int x)
{
  return x*x;
}
```

Listing 1: Source code.

```
(func (param i32) (result i32)
    local.get 0
    local.get 0
    i32.mul
)
```

Listing 2: `wat` format.

```
20 00 | local.get 0
20 00 | local.get 0
6c    | i32.mul
0b    | end
```

Listing 3: `wasm` format.

Figure 1: Example of a function compiled to Wasm.

only inside the current function. Multi-way branches can only target blocks that are statically designated in a branch table. Unrestricted gotos or jumps to arbitrary addresses are not possible. In particular, one cannot execute data in memory as bytecode instructions. Thus, attacks like shellcode injection or abuse of indirect jumps are not possible.

The execution semantics guarantee the safety of direct function calls through the use of explicit function section indexes, and returns through a protected call stack. The type signature of indirect function calls is checked at runtime, effectively implementing coarse-grained type-based control-flow integrity for indirect calls. Additionally, the LLVM compiler infrastructure includes a built-in implementation of fine-grained control flow integrity, which has been extended to support the WebAssembly target [16].

2.2.2 Vulnerabilities

Vulnerabilities in memory-unsafe languages, like C and C++, can translate to vulnerabilities in WebAssembly binaries [48]. Many vulnerabilities which, due to common mitigations, are no longer exploitable in native binaries, are exploitable in WebAssembly. Moreover, WebAssembly enables unique attacks, such as overwriting constant data or manipulating the heap using a stack overflow. Additionally, the Emscripten API [7] allows developers to access the DOM, which can be exploited to inject malicious input in the form of Cross Site Scripting (XSS) attacks [58]. Unfortunately, two-thirds of WebAssembly binaries are compiled from memory-unsafe languages [41], and these attacks have been shown to be practical in real-world scenarios [48].

WebAssembly vulnerabilities have been known to be exploited in the wild. Fastly, a cloud platform that offers edge computing services, experienced a 45-minute disruption on June 8th, 2021, when a WebAssembly binary was deployed [5]. Vulnerabilities in WebAssembly smart contracts have also been exploited. For instance, random number generation vulnerabilities have resulted in the loss of around 170,000 EOS tokens [63]. The fake EOS transfer vulnerability in the EOSCast smart contract has resulted in the loss of around 60,000 EOS tokens [42]. The forged transfer notification vulnerability in EOSBet has resulted in the loss of 140,000 EOS tokens [42]. Based on the average price of EOS tokens at the time of the attacks, the total loss from these three vulnerabilities was around $1.9 million. Additionally, around 25% of WebAssembly smart contracts have been found to be vulnerable [40].

2.2.3 Cryptojacking

Cryptojacking, also known as drive-by mining, involves using a website visitor's hardware resources for mining cryptocurrencies without their consent. Previously, cryptojacking was implemented using JavaScript. However, in recent years WebAssembly has been utilized due to its computational efficiency. The year after WebAssembly was released, there was a 459% increase in cryptojacking [24]. The following year, researchers found that over 50% of all sites using WebAssembly were using it for cryptojacking [60]. To counter this trend, researchers developed several static and dynamic detection methods for identifying WebAssembly-based cryptojacking.

3 Related Work

This section discusses related work. Specifically, related studies are presented and the differences between those studies and our paper are discussed.

In a similar vein to this paper, Kim et al. [46] survey the various techniques and methods for WebAssembly binary security. However, their focus is on general security techniques for WebAssembly, while our paper focuses specifically on analysis techniques for WebAssembly. We both discuss cryptojacking detection and vulnerability detection for WebAssembly, but we go further by also examining vulnerability analysis for WebAssembly smart contracts. Additionally, we use different classification systems and performance metrics.

Tekiner et al. [77] focus on surveying cryptojacking detection techniques by strictly evaluating and comparing state-of-the-art methods. In contrast, our paper examines analysis techniques for WebAssembly, including cryptojacking detection, vulnerability analysis for WebAssembly binaries, and vulnerability analysis for WebAssembly smart contracts. We also use different classification systems and performance metrics.

Romano et al. [67] investigate bugs in WebAssembly compilers, specifically examining the Emscripten [14], Assem-

blyScript [9], and WebAssembly-Bindgen [69] compilers. They discover bugs in the Emscripten compiler that could potentially cause significant security issues. Our work, on the other hand, focuses on security in WebAssembly binaries using analysis techniques, rather than examining the security of the compilers themselves.

4 Analysis Techniques for WebAssembly

This section presents the results of our literature review on analysis techniques for WebAssembly. The techniques can be broadly classified into three categories:

1. Detecting malicious WebAssembly binaries (§4.1)
2. Detecting vulnerabilities in WebAssembly binaries (§4.2)
3. Detecting vulnerabilities in WebAssembly smart contracts (§4.3)

The analysis techniques can be further classified as either static, dynamic, or hybrid methods. Some techniques are static-based, meaning they analyze the WebAssembly bytecode or intermediate representation without executing the binary. Other techniques are dynamic-based, meaning they analyze the behavior of the WebAssembly binary as it is being executed. Finally, some techniques are hybrid-based, meaning they combine both static and dynamic analysis to detect potential security issues. The classification is summarized in Table 1.

The performance of the various analysis techniques has been evaluated using a set of metrics that measure their effectiveness and efficiency. These metrics include precision, recall, and F_1-score. Precision measures the proportion of retrieved items that are relevant, while recall measures the proportion of relevant items that are retrieved. A high number of false positives will decrease the precision, while a high number of false negatives will decrease the recall. The F_1 score is the harmonic mean of precision and recall, and provides a way to combine these two metrics into a single value.

These metrics are used instead of accuracy because they are better suited for evaluating the performance of analysis techniques in the presence of imbalanced datasets, which has been common in the literature. In addition to these metrics, the performance of static-based methods has been evaluated using detection time, while the performance of dynamic-based methods has been evaluated using runtime overhead. These metrics provide a way to compare the different analysis techniques and assess their relative strengths and weaknesses.

4.1 Detecting Malicious WebAssembly Binaries

As previously mentioned, WebAssembly has been used by adversaries for malicious purposes, like cryptojacking (§2.2.3). To protect against such attacks, several detection techniques have been proposed. In this section, we will review these techniques and evaluate their performance.

Techniques based on static analysis (§4.1.1) and dynamic analysis (§4.1.2) are discussed in the following sections. Additionally, a comparative analysis of the detection techniques is presented (§4.1.3).

4.1.1 Static Analysis

MineSweeper MineSweeper [47] detects cryptojacking based on the presence of cryptographic functions in WebAssembly binaries. MineSweeper implements two variants: The first variant is specialized for detecting the CryptoNight algorithm [33], which is commonly used for cryptomining, while the second variant is more generic and can detect any cryptographic function that may be used for cryptomining. A cryptographic fingerprint is computed by counting the number of cryptographic operations in each function in the WebAssembly binary. In the case of the CryptoNight variant, these fingerprints are then compared with the fingerprints of the primitive components of the CryptoNight algorithm. In the generic case, a candidate function is labeled as a cryptographic function if the amount of cryptographic operations exceeds a threshold. The authors conducted experiments to validate the effectiveness of their method, achieving 100% recall and precision for both variants. However, they acknowledge a potential limitation of MineSweeper: It can produce false positives, as benign programs such as games and cryptographic libraries also use cryptographic functions.

MinerRay MinerRay [68] constructs and analyses an Inter-Procedural Control Flow Graph to detect cryptojacking. MinerRay first converts JavasScript and asm.js code into WebAssembly binaries. Then, the WebAssembly binaries are translated into an intermediate representation, from which Intra-Procedural Control Flow Graphs are constructed for each function. These Intra-Procedural Control Flow Graphs are then linked together to create an Inter-Procedural Control Flow Graph that represents the entire program. MinerRay uses the Inter-Procedural Control Flow Graph 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 cryptomining, MinerRay employs a dynamic approach that explores `onclick` events of HTML objects, which may instantiate WebAssembly cryptominers. It then checks if the WebAssembly APIs, such as `WebAssembly.instantiate`, can be invoked. Out of 901 websites with cryptominers, the authors found that only 16 websites informed users of the background cryptomining, and just three of those asked for consent before starting the mining process.

MINOS MINOS [61] uses an image-based classification deep learning approach to identify cryptojacking. First, MI-

Category	Static analysis	Dynamic analysis	Hybrid analysis
Detecting malicious WebAssembly binaries (§4.1)	MineSweeper [47]	SEISMIC [80]	
	MinerRay [68]	RAPID [66]	
	MINOS [61]	OutGuard [45]	
		MineThrottle [26]	
		CoinSpy [44]	
Detecting vulnerabilities in WebAssembly binaries (§4.2)	Wassail [74]	Szanto et al. [76]	$WASP_2$ [75]
	Wasmati [55]	TaintAssembly [35]	
	$WASP_1$ [57]	Wasabi [49]	
		Fuzzm [50]	
		WAFL [39]	
Detecting vulnerabilities in WebAssembly smart contracts (§4.3)	EVulHunter [65]	EOSFuzzer [42]	
	WANA [79]	WASAI [28]	
	EOSAFE [40]		
	EOSIOAnalyzer [51]		

Table 1: Classification of analysis techniques for WebAssembly.

NOS converts the WebAssembly binary into a grayscale image. This image is then used as input to a Convolutional Neural Network (CNN), which has been trained on a comprehensive 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. An advantage of MINOS is that it is lightweight and can detect cryptojacking in under a second. This makes it a useful tool for real-time cryptojacking detection.

4.1.2 Dynamic Analysis

SEISMIC SEISMIC [80] uses signature-matching to identify cryptojacking. It adopts an In-Line Reference Monitor (IRM) approach, which involves dynamically computing semantic features of the WebAssembly binary at runtime. To this end, an instruction counter is inserted into the global section of the WebAssembly binary for each instruction to be profiled. The semantic features of the WebAssembly binary are then computed using the aforementioned instruction counters. To identify cryptojacking, the computed semantic features of the WebAssembly binary are compared with semantic signatures of known mining binaries. This approach was found to be accurate in detecting cryptojacking, but it imposes a significant runtime overhead, which can affect the performance of the WebAssembly application.

RAPID RAPID [66] identifies cryptojacking by monitoring JavaScript API calls and system resource usage. To this end, JavaScript API usage is collected using chrome debugging features. The system resources, that is, the memory, network and the processor usage is collected by executing a Chromium instance inside a docker container and collecting the data through the docker stats API [11]. Then, a Support Vector Machine (SVM) is employed as a classification model.

OutGuard OutGuard [45] uses features related to the JavaScript runtime execution, event loads, networking, and cryptojacking libraries to detect cryptojacking. Specifically, the number of web workers and parallel tasks, the existence of WebAssembly modules, WebSockets and hashing algorithms, and the usage of PostMessage- and MessageLoop event loads are used as the feature set. These seven distinct features are used to build an SVM classification model. A limitation of this approach is that the identification of hashing algorithms is static and does not account for string obfuscation.

MineThrottle MineThrottle [26] uses the frequency distribution of instructions to detect cryptojacking. The idea is that miners execute certain instructions more frequently than benign applications, and this can be used to identify mining activity. To implement this, MineThrottle first detects potential mining-related code blocks using block-level statistical features, and then instruments each block using block-level program profiling. The effective mining speed (i.e., the instructions per cycle) of the WebAssembly program is then periodically calculated, and if it is similar to known mining programs, the program is labeled as a miner.

CoinSpy CoinSpy [44] is a method for detecting cryptojacking by monitoring compute, memory, and network usage from within the browser. The computational behavior is monitored using the JavaScript stack profiler, and memory usage is measured by monitoring the JavaScript heap and WebWorker

Scheme	Feature(s)	Classifier	Dataset		Performance			
			Source	Samples	Precision	Recall	F_1	DT
MineSweeper (2018)	WebAssembly code	Matching or threshold	Alexa 1M	748	100%	100%	100%	-
MinerRay (2020)	WebAssembly code	ICFG	Alexa 1.2M	3825	99%	100%	99%	1.9s
MINOS (2021)	WebAssembly code	CNN	Tranco 100K, PublicWWW	682	93%	97%	95%	0.0259s

Abbreviations: Detection Time (DT).

Table 2: Data for static detection techniques for identifying malicious WebAssembly binaries.

threads. Network usage is tracked by summing the bytes from all in-flight requests at each millisecond. The key observation used for cryptojacking detection is that compute and memory usage increase significantly when the Proof of Work (PoW) algorithm is executing, and that network usage only increases when the processor is in an idle state. Using these features, a CNN classification model was constructed. The authors argue that CoinSpy should be able to detect future cryptomining algorithms that other dynamic detectors will miss due to their specificity.

4.1.3 Comparative Analysis

This section presents the comparative analysis of the detection techniques outlined in the above sections. The results of the analysis are summarized in Table 2 and Table 3.

Dataset Most detection techniques have been evaluated using websites collected from the wild, with the Alexa sites being the most commonly used. Only SEISMIC evaluated their method using a curated list of binaries. Most schemes used a sufficient number of samples, but there were some exceptions, such as SEISMIC, MineThrottle, and MINOS, which had a smaller number of samples, potentially affecting the validity of their results.

Performance The performance of the detection techniques was evaluated using metrics such as precision, recall, F_1 score, overhead, and detection time. Among the static-based methods, MineSweeper and MinerRay had the highest $F1_1$ scores, while MINOS had the lowest. However, MINOS also had the fastest detection time, making it suited for real-time cryptojacking detection. Among the dynamic-based methods, MineThrottle, Outguard, and SEISMIC had the highest F_1 scores. However, SEISMIC also had the highest overhead. In contrast, MineThrottle and Outguard had negligible overhead.

4.2 Detecting Vulnerabilities in WebAssembly Binaries

Although WebAssembly was designed with security in mind, vulnerabilities still exist (§2.2.2). As a result, various techniques for detecting vulnerabilities in WebAssembly binaries have been proposed. This section presents these techniques and discusses their versatility, which is determined by factors such as compatibility with different runtimes, support for the WASI, and whether they require high-level source code for analysis.

Techniques based on static analysis (§4.2.1), dynamic analysis (§4.2.2), and hybrid analysis (§4.2.3) are discussed in the following sections. Additionally, a comparative analysis of the detection techniques is presented (§4.2.4).

4.2.1 Static analysis

Wassail Wassail [74] was the first static analysis method for detecting vulnerabilities in WebAssembly binaries. It uses a compositional, summary-based analysis approach that strictly focuses on information flow. For each WebAssembly function, it computes a summary that describes how information flows within that function, and these summaries are then used during the subsequent analysis of function calls. The information flow analysis is expressed as a data flow analysis on a Control Flow Graph (CFG), and the information flow of the entire program is approximated by composing the function summaries. The authors claim that similar approaches have been shown to scale well [27, 43], but the scalability of Wassail has not been evaluated.

Wasmati Wasmati [55] detects vulnerabilities in WebAssembly binaries by constructing a Code Property Graph (CPG). Vulnerabilities are detected by searching for specific patterns in the CPG sub-graphs, which includes the execution order, execution path, data dependencies, and control flows.

Scheme	Feature(s)	Classifier	Dataset		Performance			
			Source	Samples	Precision	Recall	F_1	Overhead
SEISMIC (2018)	WebAssembly code, instruction count obtained at runtime	Matching	Asteroids, A-Star, Tanks, Bullet(1000), CoinHive_v0, CoinHive, Basic4GL, HushMiner, CreaturePack FunkyKarts, NFWebMiner, YAZECMiner	12	96%	100%	98%	100%
RAPID (2018)	JavaScript API calls, memory, processor and network usage	SVM	Alexa 330K	71450	97%	96%	96%	9-40%
OutGuard (2019)	Parallel tasks, WebAssembly, hashing algorithms, WebSockets, PostMessage event load, MessageLoop event load	SVM, or RF	Alexa 1M, Alexa 600K	29700	99%	97%	98%	2%
CoinSpy (2020)	JavaScript stack execution time, JavaScript heap, network usage	CNN	Alexa 1M, Alexa 100K, PublicWWW	2000	Accuracy: 97%			0%
MineThrottle (2020)	WebAssembly code, processor usage	Matching	Alexa 1M	659	100%	98%	99%	0%

Table 3: Data for dynamic detection techniques for identifying malicious WebAssembly binaries.

An issue with this approach is that the number of nodes in the CPG grows rapidly because the target of indirect calls cannot be determined statically. To address this, Wasmati optimizes the CPG generation process by adding additional annotations, caching intermediate results, and using efficient graph traversal algorithms. As a result, the authors found that constructing the CPG only took an average of 58 seconds per binary. They also found that Wasmati was able to effectively find vulnerabilities in WebAssembly binaries while providing a low false positive rate.

WASP$_1$ WASP$_1$ [57] is a concolic execution engine for WebAssembly modules that can be used for uncovering vulnerabilities and bugs. Concolic execution, which combines concrete execution with symbolic execution and explores one execution path at a time, is employed to explore all feasible paths of the program. Specifically, symbolic execution is used to generate concrete inputs for exploring multiple execution paths, with the goal of maximizing code coverage. To demonstrate the feasibility of uncovering vulnerabilities, the authors constructed WASP-C, a symbolic execution framework for testing C programs using WASP$_1$. WASP-C takes a C program as input, annotates it, compiles it to WebAssembly, and analyzes it using WASP$_1$. They found that WASP-C was effective at uncovering bugs and vulnerabilities. However, a limitation of WASP-C is that it requires high-level source code to uncover vulnerabilities, meaning it can only be used to analyze open-source programs.

4.2.2 Dynamic Analysis

Szanto et al. Szanto et al. [76] proposed a taint-tracking technique for detecting vulnerabilities in WebAssembly binaries. They developed a VM that runs in native JavaScript and implemented a taint tracking system that allows the user to monitor the flow of sensitive data through the execution of the WebAssembly binary. To this end, they allocate a tainted label for each allocable byte in the memory section and each variable on the stack. This method allows for taint tracking without modifying the structure of the WebAssembly binary.

Type	Scheme	Technique	Runtime	Binary-only	WASI support	Overhead
Static	Wassail (2020)	Compositional information flow	-	✓	-	-
	Wasmati (2022)	CPG	-	✓	-	-
	$WASP_1$ (2022)	Concolic execution	-	✗	-	-
Dynamic	Szanto et al. (2018)	Taint tracking	WebAssembly VM (Custom)	✓	✗	100%
	TaintAssembly (2018)	Taint tracking	V8 engine (Modified)	✓	✓	5-12%
	Wasabi (2019)	Binary instrumentation	Any runtime	✓	✓	2-163%
	Fuzzm (2021)	Fuzzing	Any runtime w/ WASI-support	✓	✓	5-6%
	WAFL (2021)	Fuzzing	WAVM (Modified)	✓	✓	-
Hybrid	$WASP_2$ (2021)	Known vulnerabilities	-	✓	✓	-

Table 4: Data for dynamic detection techniques for identifying malicious WebAssembly binaries.

The authors found that the runtime overhead of this method scales mostly linearly, with an overhead of up to 100%.

TaintAssembly TaintAssembly [35] is another technique that uses taint-tracking to detect vulnerabilities in WebAssembly binaries. Unlike Szanto et al., who developed their own VM, TaintAssembly implemented taint-tracking by modifying the V8 JavaScript engine used in Google Chrome and Node.js [62]. TaintAssembly implements basic taint-tracking functionality for variables of type `i32`, `i64`, `f32`, `f64`, as well as tainting in linear memory and a probabilistic variant of taint. However, unlike Szanto et al.'s approach, the structure of the WebAssembly module must be modified before taint labels can be set for all variables. TaintAssembly was able to achieve a runtime overhead of only 5-10%, which is far less than Szanto et al.'s approach.

Wasabi Wasabi [49] is a general-purpose framework for dynamically analyzing WebAssembly binaries. To this end, Wasabi performs binary instrumentation. Specifically, it inserts calls to analysis functions written in JavaScript into the WebAssembly binary. Then, instruction counting, call graph extraction, memory access tracing, and taint analysis can be performed at runtime. Wasabi also allows for selective instructions; that is, it only instruments instructions that are relevant for a particular analysis. The authors found the runtime overhead to vary between 2% and 163%, depending on the application and instructions being analyzed.

Fuzzm Fuzzm [50] is a binary-only fuzzer for WebAssembly that uses the popular AFL [37] framework. Native AFL compiles applications from source code and inserts code to track path coverage. However, since Fuzzm is a binary-only fuzzer, it does not have access to the source code. To provide coverage information for the AFL fuzzer, Fuzzm uses static binary instrumentation to insert code at all branches, generating AFL-compatible coverage information. The authors found that Fuzzm is effective and imposes a low runtime overhead. Additionally, its implementation is not tied to a specific runtime. Fuzzm also implements a canary-based protection mechanism to prevent memory corruption vulnerabilities.

WAFL WAFL [39] is also a binary-only fuzzer for WebAssembly. It uses the AFL++ [34] framework, a community-driven fork of AFL. To generate coverage for the AFL++ fuzzer, they implement a set of patches to the WAVM [6] runtime. The WAVM runtime uses Ahead-of-time (AOT) compilation, and WAFL also add lightweight VM snapshots. This makes WAFL performant, in some cases even outperforming native AFL x86-64 harnesses compiled from source. However, WAFL is inherently tied to the WAVM runtime, which limits its potential use cases.

4.2.3 Hybrid Analysis

WASP$_2$ WASP$_2$ [75] detects vulnerabilities in WebAssembly binaries based on known vulnerabilities. It does this by analyzing static and dynamic features of the WebAssembly binary, and compares this with known vulnerabilities. Specifically, WASP$_2$ trains a deep learning vulnerability classification model by mapping static features of known vulnerable binaries in x86 or ARM, to static features in the corresponding WebAssembly binary representation. Then, the model is used to statically analyze the WebAssembly binary. Finally, the identified vulnerable subroutines are dynamically analyzed using Wasabi [49]. The authors found that WASP$_2$ is able to accurately find known vulnerabilities in WebAssembly binaries.

4.2.4 Comparative Analysis

This section presents the comparative analysis of the detection techniques outlined in the above sections. The results from the analysis are summarized in Table 4.

Runtime Compatibility The versatility of the proposed detection techniques is determined by their runtime compatibility, WASI support, and whether they require high-level source code for their analysis. Some schemes, like TaintAssembly, are inherently tied to a specific runtime, which limits their usefulness. Other schemes, like Wasabi, are not tied to any specific runtime and can be applied more generally. Additionally, schemes that do not support WASI, like Szanto et al.'s method, are fundamentally limited since they cannot analyze most WebAssembly binaries used on servers or embedded devices. Finally, schemes that require high-level source code for analysis, like WASP$_1$, are limited to the analysis of open-source projects.

Overhead The overhead for dynamic detection methods varies greatly. Wasabi and Szanto et al. have the highest overhead, reaching up to 163% and 100%, respectively. Szanto et al.'s method have a constant overhead, while Wasabi's overhead varies depending on the type and number of instructions being analyzed. This means that Wasabi's overhead can be as low as 2% in practice. TaintAssembly and Fuzzm have the lowest overhead, ranging from 5-12% and 5-6%, respectively.

4.3 Detecting Vulnerabilities in WebAssembly Smart Contracts

Several vulnerabilities have been discovered in WebAssembly smart contracts, leading to significant financial loss (§2.2.2). As a result, several techniques for detecting such vulnerabilities have been developed. This section presents these techniques, their capabilities, and performance.

Techniques based on static analysis (§4.3.1) and dynamic analysis (§4.3.2) are discussed in the following sections. Additionally, a comparative analysis of the detection techniques is presented (§4.3.3).

4.3.1 Static Analysis

EVulHunter EVulHunter [65] was the first static analysis tool designed to detect vulnerabilities in EOSIO smart contracts. It uses the open-source analysis framework Octopus [36] to construct a CFG of the smart contract. The CFG is then traversed to detect vulnerabilities based on predefined patterns. Although EVulHunter is effective at detecting fake notification vulnerabilities, it has low precision when detecting fake EOS transfers. The authors believe this is due to the limitations of using predefined patterns and suggest that more advanced analysis techniques, such as symbolic execution, is necessary.

WANA WANA [79] uses symbolic execution and a set of test oracles to detect vulnerabilities in smart contracts. It is cross-platform, meaning it can detect vulnerabilities in both EOSIO and Ethereum smart contracts. First, Ethereum smart contracts, which are written in Solidity, are converted to Ewasm (Ethereum flavored WebAssembly) using the SOLL [71] compiler. Then, the symbolic execution engine traverses the paths of the WebAssembly binary. WANA performs vulnerability analysis based on the data collected during symbolic execution using the proposed test oracles. Unlike EVulHunter, WANA can also detect blockinfo dependency vulnerabilities and detect fake EOS transfer vulnerabilities effectively.

EOSAFE EOSAFE [40] also uses symbolic execution to detect vulnerabilities in EOSIO smart contracts. Unlike WANA, EOSAFE addresses the problem of path explosion, where the number of feasible paths in a program grows exponentially with the program size. To mitigate this issue, EOSAFE allows users to set call depth and timeout parameters that are used when symbolically executing the program. Additionally, during vulnerability detection, EOSAFE first identifies valuable functions (i.e., functions that have the ability to invoke actions or change on-chain state) and only analyzes those. This approach allows it to accurately detect more vulnerabilities than previous methods.

EOSIOAnalyzer EOSIOAnalyzer [51] detects vulnerabilities in EOSIO smart contracts by analyzing the ICFG of the program. The ICFG is the combination of the CFG and the

Type	Scheme	Technique	Vulnerability detection					Performance			
			FE	FN	BD	RB	MAV	Precision	Recall	F_1	DT
Static	EVulHunter (2019)	CFG	✓	✓	✗	✗	✗	89%	100%	93%	1-3s
	WANA (2020)	Symbolic execution	✓	✓	✓	✗	✗	100%	100%	100%	0.21s
	EOSAFE (2021)	Symbolic execution	✓	✓	✗	✓	✓	100%	96%	98%	-
	EOSIOAnalyzer (2022)	ICFG	✓	✓	✓	✗	✗	93%	100%	96%	7.6s
Dynamic	EOSFuzzer (2020)	Fuzzing	✓	✓	✓	✗	✗	88%	88%	88%	-
	WASAI (2022)	Concolic fuzzing	✓	✓	✓	✓	✓	100%	98%	99%	-

Abbreviations: Fake EOS (FE), Fake Notification (FN), Blockinfo Dependency (BD), Rollback (RB), Missing Authorization Verification (MAV), and Detection Time (DT).

Table 5: Data for detecting vulnerabilities in WebAssembly smart contracts.

Call Graph (CG) of the program, allowing EOSIOAnalyzer to analyze data propagation relationships between functions when they call each other. After the ICFG is constructed, the WebAssembly code is translated into a high-level intermediate representation. Then, EOSIOAnalyzer applies a data flow analysis algorithm to determine the data propagation relationships between functions. Finally, it identifies suspicious functions and further analyzes their complete execution paths. To address the issue of path explosion, EOSIOAnalyzer implements a call depth threshold.

4.3.2 Dynamic Analysis

EOSFuzzer EOSFuzzer [42] uses black-box fuzzing to detect vulnerabilities in EOSIO smart contracts. To this end, EOSFuzzer first performs static analysis on the WebAssembly code and Application Binary Interface (ABI). The results of this analysis are then used to generate fuzzing inputs, which are applied to the smart contract through the Cleos [10] command-line client. Finally, EOSFuzzer performs vulnerability analysis based on test oracles. Although EOSFuzzer was found to be relatively efficient, it has the lowest precision and recall of all methods proposed. Additionally, EOSFuzzer uses random seeds for fuzzing, resulting in low code coverage.

WASAI WASAI [28] uses concolic fuzzing to detect vulnerabilities in EOSIO smart contracts. To address the weaknesses of EOSFuzzer, it strategically generates seeds to aid the fuzzing in exploring as many feasible paths as possible. This is done by performing symbolic execution to feedback the seed mutation. This results in double the code coverage than EOSFuzzer. WASAI was able to detect the most vulnerabilities out of all the proposed techniques. It additionally has high precision and recall, and was found to be resilient to code obfuscation.

4.3.3 Comparative Analysis

This section presents the comparative analysis of the detection techniques outlined in the above sections. The results from the analysis are summarized in Table 5.

Vulnerability Detection The proposed detection techniques are able to detect different types of vulnerabilities. EVulHunter is only able to detect two out of five types of vulnerabilities, while WANA, EOSIOAnalyzer, and EOSFuzzer are able to detect three out of five types. EOSAFE is able to detect four out of five vulnerabilities. WASAI is the only method that is able to detect all five vulnerabilities, making it the most effective detection method.

Performance WANA and WASAI have the highest F_1-score among the proposed detection techniques, with 100% and 99%, respectively. EOSFuzzer and EVulHunter have the lowest F_1 scores, with 88% and 93%, respectively. In terms of detection time, EOSIOAnalyzer has the slowest detection time at 7.6 seconds. In contrast, WANA has a detection time of only 0.21 seconds while providing high precision and recall.

5 Discussion

This section presents the results of the literature review. It begins by summarizing the key findings, followed by a discussion of the limitations of current analysis techniques. Finally, future research directions are proposed.

5.1 Key Findings

Methods based on static analysis use techniques such as signature matching and symbolic execution which do not require the program to be executed. This allows them to impose a low overhead but can also result in lower accuracy. For example, MINOS has the fastest detection time at under one second but also has the lowest F_1-score of all cryptojacking detection methods. Methods that rely solely on signature or keyword matching can be easily bypassed through the use of obfuscation techniques [25, 80]. Additionally, methods that rely on semantic execution may be limited by the path explosion problem.

Methods based on dynamic analysis execute the program in a controlled environment to extract behavioral features that can be used for further analysis. To do this, various techniques such as taint tracking, binary instrumentation, fuzzing, and monitoring system resources have been proposed. This typically results in higher overhead but also better performance than static-based methods. For example, Wasabi is able to perform heavy-weight dynamic analysis through binary instrumentation, but it might also incur a high overhead. Since dynamic analysis is based on behavioral features, it is less susceptible to evasion through obfuscation techniques. However, it can still be bypassed in some cases, such as by throttling processor usage.

Static and dynamic techniques are not mutually exclusive, but complementary techniques. A hybrid approach can leverage the low overhead of static techniques and the high accuracy of dynamic techniques. In such an approach, static techniques can be used to identify candidate functions, and dynamic techniques can then be employed to analyze them accurately. Currently, only $WASP_2$ employs such a hybrid approach.

5.2 Limitations

The evaluation strategies for each detection method differ substantially. Some methods are evaluated using imbalanced datasets, while others use balanced datasets. Additionally, the sample sizes used for evaluation also differ between the detection methods. To address these variations, we have used precision, recall, and F_1 as performance metrics instead of accuracy. However, variations in evaluation strategies can still impact the validity of the results. For example, EVulHunter reported an F_1-score of 93%, but other studies found its F_1 score to be 17% and 23% [42, 51]. However, using the results from these complementary studies may further threaten the validity of the results, as they may contain implementation bugs.

Many cryptojacking detection methods do not distinguish between cryptomining and cryptojacking. Cryptomining refers to the use of a user's resources to mine cryptocurrency with the user's explicit consent. This has been used as an alternative revenue source by organizations such as UNICEF [52]. Cryptojacking, on the other hand, refers to the use of a user's resources to mine cryptocurrency without their explicit consent. Currently, only MinerRay is able to differentiate between these two activities. This differentiation may result in a 1-2% increase in false positives [68].

The datasets used for evaluation can impact the results of the evaluation. Cryptojacking websites often modify or move their scripts to different domains to avoid being blacklisted. Additionally, the CoinHive shutdown [41, 78] resulted in a decrease in cryptojacking activity. As a result, the accuracy of cryptojacking detection methods may vary depending on when the dataset was collected.

5.3 Research Directions

Generally, the proposed WebAssembly analysis techniques are focused on the web environment. As WebAssembly is being extended for use beyond the web, current analysis techniques do not cover all possible use cases. There have been studies on the use of WebAssembly in non-web environments [73], but few have specifically focused on its security in these contexts. Further research addressing the security of WebAssembly in non-web environments is needed.

The proposed detection techniques for detecting malicious WebAssembly binaries are biased towards cryptojacking. WebAssembly can also be used for other malicious purposes, like tech support scams, browser exploits, and script-based keyloggers [54]. Currently, there are no methods for detecting these types of malicious uses of WebAssembly. Further research is encouraged in this direction.

The proposed methods for detecting cryptojacking can be circumvented through code obfuscation, which has previously rendered static detection methods obsolete [72]. Obfuscation of WebAssembly code is common on the web [41, 47]. However, only one preliminary study [25] has investigated the feasibility of obfuscation for WebAssembly, and the researchers only evaluated it using one static detection technique. The effects on dynamic detection techniques were not

explored. Additionally, the study used a small dataset, potentially undermining the validity of the results. Some authors of cryptojacking detection techniques argue that obfuscation is impractical due to the added runtime overhead and the resulting decrease in revenue from reduced hash rates. However, the effects of obfuscated WebAssembly code on runtime and hash rates have not been studied. More research in this area is needed.

The prevalence of WebAssembly-based cryptojacking on the web is unclear. There have been two studies on this topic, one by Musch et al. [60] in 2018 and the other by Hilbig et al. [41] in 2021. Musch et al. found that over 50% of sites using WebAssembly were doing so for cryptojacking, while Hilbig et al. found that this number had been marginalized to 1%. This decrease was attributed to the shutdown of CoinHive, which is supported by other studies [78]. However, even after the shutdown of CoinHive, other studies have found the prevalence of WebAssembly-based cryptojacking to be as high as 10% [61]. Moreover, Hilbig et al. used VirusTotal [19] for detecting cryptojacking, which has been proven to be easily bypassed through code obfuscation [80]. Therefore, the results of this study may be inaccurate due to false negatives. Further research in this area is needed.

6 Conclusion

In this paper, we conducted a comprehensive review of analysis techniques for WebAssembly. To this end, we constructed a taxonomical classification and applied it to analysis techniques proposed in the literature. We classified the techniques into three categories: Detecting malicious WebAssembly binaries, detecting vulnerabilities in WebAssembly binaries, and detecting vulnerabilities in WebAssembly smart contracts. We analyzed these techniques using quantitative data and discussed their strengths and weaknesses. Then, key findings and limitations were presented. Specifically, we found that static methods have low overhead but lower accuracy, while dynamic analysis has higher overhead but higher accuracy. We also identified potential areas for future research, including the security of WebAssembly in non-web environments, analysis techniques for malicious WebAssembly binaries, the feasibility of obfuscating WebAssembly code, and the prevalence of WebAssembly-based cryptojacking on the web. This paper provides a valuable contribution to the field by offering a comprehensive understanding of current analysis techniques for WebAssembly, including their use cases and limitations, as well as suggestions for future research.

References

[1] Native Client - Chrome Developers. https://developer.chrome.com/docs/native-client, January 1980. [Accessed 4 Nov. 2022].

[2] 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].

[3] EOS Virtual Machine: A High-Performance Blockchain WebAssembly Interpreter – EOSIO. https://eos.io/news/eos-virtual-machine-a-high-performance-blockchain-webassembly-interpreter, January 2021. [Accessed 9. Nov. 2022].

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

[5] Summary of June 8 outage. https://www.fastly.com/blog/summary-of-june-8-outage, June 2021. [Accessed 29 Nov. 2022].

[6] WAVM. https://wavm.github.io, October 2021. [Accessed 3 Nov. 2022].

[7] API Reference — Emscripten 3.1.26-git (dev) documentation. https://emscripten.org/docs/api_reference/index.html, December 2022. [Accessed 1 Dec. 2022].

[8] asm.js - Game development | MDN. https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js?source=post_page, November 2022. [Accessed 4 Nov. 2022].

[9] AssemblyScript. https://www.assemblyscript.org, November 2022. [Accessed 24 Nov. 2022].

[10] Cleos – EOSIO. https://eos.io/for-developers/build/cleos, November 2022. [Accessed 29 Nov. 2022].

[11] Develop with Docker Engine API. https://docs.docker.com/engine/api, November 2022. [Accessed 3. Nov. 2022].

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

[13] Introduction to Portable Native Client. https://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client, December 2022. [Accessed 2 Dec. 2022].

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

[15] Same Origin Policy - Web Security. https://www.w3.org/Security/wiki/Same_Origin_Policy, November 2022. [Accessed 26 Nov. 2022].

[16] Security - WebAssembly. https://webassembly.org/docs/security/#users, November 2022. [Accessed 3 Nov. 2022].

[17] Usage Statistics of JavaScript as Client-side Programming Language on Websites, December 2022. https://w3techs.com/technologies/details/cp-javascript, November 2022. [Accessed 4 Nov. 2022].

[18] Using the WebAssembly JavaScript API - WebAssembly | MDN. https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API, November 2022. [Accessed 29 Nov. 2022].

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

[20] WASI |. https://wasi.dev, June 2022. [Accessed 25 Nov. 2022].

[21] wasm-pack. https://rustwasm.github.io/wasm-pack, September 2022. [Accessed 23 Nov. 2022].

[22] WebAssembly | Can I use... Support tables for HTML5, CSS3, etc. https://caniuse.com/wasm, November 2022. [Accessed 12 Nov. 2022].

[23] What is a Smart Contract? | NEAR Documentation. https://docs.near.org/develop/contracts/whatisacontract, November 2022. [Accessed 11 Nov. 2022].

[24] 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].

[25] 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.

[26] 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.

[27] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[28] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 703–715, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Contributors to Wikimedia projects. JavaScript - Wikipedia. https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1126827786, November 2001. [Accessed 3 Dec. 2022].

[30] Contributors to Wikimedia projects. ActiveX - Wikipedia. https://en.wikipedia.org/w/index.php?title=ActiveX&oldid=1102963222, August 2022. [Accessed 4 Nov. 2022].

[31] Contributors to Wikimedia projects. Adobe Flash - Wikipedia. https://en.wikipedia.org/w/index.php?title=Adobe_Flash&oldid=1126708043, December 2022. [Accessed 2 Dec. 2022].

[32] Contributors to Wikimedia projects. Java (programming language) - Wikipedia. https://en.wikipedia.org/w/index.php?title=Java_(programming_language)&oldid=1126888277, December 2022. [Accessed 2 Dec. 2022].

[33] Coolstory. CryptoNight – CryptoNote Protocol – BitcoinWiki. *BitcoinWiki*, December 2018.

[34] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[35] William Fu, Raymond Lin, and Daniel Inge. Taintassembly: Taint-based information flow control tracking for webassembly. *arXiv preprint arXiv:1802.01050*, 2018.

[36] FuzzingLabs. octopus. https://github.com/FuzzingLabs/octopus, November 2022. [Accessed 5 Nov. 2022].

[37] Google. AFL. https://github.com/google/AFL, November 2022. [Accessed 3 Nov. 2022].

[38] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 185–200, 2017.

[39] Keno Haßler and Dominik Maier. WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots. In *Reversing and Offensive-oriented Trends Symposium*, pages 23–30, 2021.

[40] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. EOSAFE: Security analysis of EOSIO smart contracts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1271–1288. USENIX Association, August 2021.

[41] 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.

[42] Yuhe Huang, Bo Jiang, and W. K. Chan. EOSFuzzer: Fuzzing EOSIO smart contracts for vulnerability detection. In *12th Asia-Pacific Symposium on Internetware*. ACM, nov 2020.

[43] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In *International Static Analysis Symposium*, pages 243–262. Springer, 2018.

[44] Conor Kelton, Aruna Balasubramanian, Ramya Raghavendra, and Mudhakar Srivatsa. Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy. In *Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web*. Internet Society, 2020.

[45] 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.

[46] Minseo Kim, Hyerean Jang, and Youngjoo Shin. Avengers, Assemble! Survey of WebAssembly Security Solutions. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 543–553. IEEE, jul 2022.

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

[48] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.

[49] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1045–1058, 2019.

[50] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly. https://arxiv.org/abs/2110.15433, 2021.

[51] Wenyuan Li, Jiahao He, Gansen Zhao, Jinji Yang, Shuangyin Li, Ruilin Lai, Ping Li, Hua Tang, Haoyu Luo, and Ziheng Zhou. EOSIOAnalyzer: An effective static analysis vulnerability detection framework for EOSIO smart contracts. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, jun 2022.

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

[53] 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.

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

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

[56] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. WebAssembly modules as lightweight containers for liquid IoT applications. In *International Conference on Web Engineering*, pages 328–336. Springer, 2021.

[57] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. Concolic Execution for WebAssembly. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. WASP.

[58] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security chasms of wasm. *NCC Group Whitepaper*, 2018.

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

[60] 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.

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

[62] Node.js. Node.js. https://nodejs.org/en, November 2022. [Accessed 27 Nov. 2022].

[63] PeckShield. Defeating EOS Gambling Games: The Tech Behind Random Number Loophole. *Medium*, November 2018. [Accessed 12 Nov 2022].

[64] 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.

[65] Lijin Quan, Lei Wu, and Haoyu Wang. EVulHunter: Detecting Fake Transfer Vulnerabilities for EOSIO's Smart Contracts at Webassembly-level. *arXiv preprint arXiv:1906.10362*, June 2019.

[66] 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.

[67] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. An Empirical Study of Bugs in WebAssembly Compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 42–54, 2021.

[68] 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.

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

[70] 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.

[71] second state. SOLL. https://github.com/second-state/soll, November 2022. [Accessed 5. Nov. 2022].

[72] Jagsir Singh and Jaswinder Singh. Challenge of malware analysis: malware obfuscation techniques. *International Journal of Information Security Science*, 7(3):100–110, 2018.

[73] Benedikt Spies and Markus Mock. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*, pages 1–10, 2021.

[74] Quentin Stiévenart and Coen De Roover. Compositional information flow analysis for webassembly programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 13–24. IEEE, 2020. WASSAIL.

[75] Pengfei Sun, Luis Garcia, Yi Han, Saman Zonouz, and Yao Zhao. Poster: Known Vulnerability Detection for WebAssembly Binaries. 2021.

[76] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint tracking for WebAssembly. *arXiv preprint arXiv:1807.08349*, 2018. Dynamic or static?

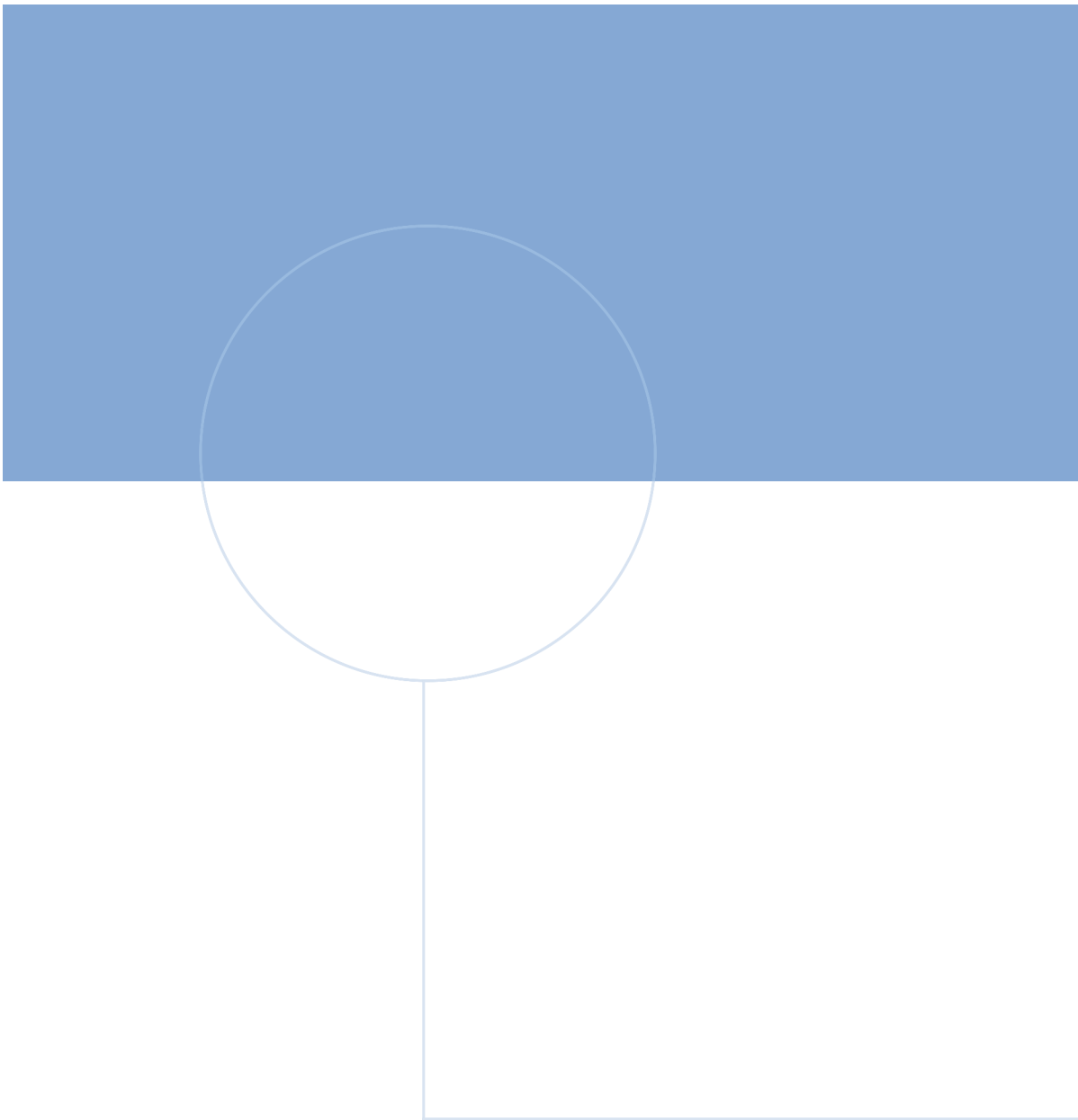
[77] Ege Tekiner, Abbas Acar, A Selcuk Uluagac, Engin Kirda, and Ali Aydin Selcuk. SoK: Cryptojacking Malware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 120–139. IEEE, 2021.

[78] Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. Is cryptojacking dead after coinhive shutdown? In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, pages 385–389. IEEE, 2020.

[79] Dong Wang, Bo Jiang, and W. K. Chan. WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection. https://arxiv.org/abs/2007.15510, 2020.

[80] 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.

[81] WebAssembly. wabt. https://github.com/WebAssembly/wabt, November 2022. [Accessed 26 Nov. 2022].



NTNU

Norwegian University of
Science and Technology