WASMIXER: Binary Obfuscation for WebAssembly

SHANGTONG CAO*, Beijing University of Posts and Telecommunications, China
NINGYU HE*, Key Lab on HCST (MOE), Peking University, China
YAO GUO, Key Lab on HCST (MOE), Peking University, China
HAOYU WANG, Huazhong University of Science and Technology, China

WebAssembly (Wasm) is an emerging binary format that draws great attention from our community. However, Wasm binaries are weakly protected, as they can be read, edited, and manipulated by adversaries using either the officially provided readable text format (i.e., wat) or some advanced binary analysis tools. Reverse engineering of Wasm binaries is often used for nefarious intentions, e.g., identifying and exploiting both classic vulnerabilities and Wasm specific vulnerabilities exposed in the binaries. However, no Wasm-specific obfuscator is available in our community to secure the Wasm binaries. To fill the gap, in this paper, we present WASMIXER, the first general-purpose Wasm binary obfuscator, enforcing data-level (string literals and function names) and code-level (control flow and instructions) obfuscation for Wasm binaries. We propose a series of key techniques to overcome challenges during Wasm binary rewriting, including an on-demand decryption method to minimize the impact brought by decrypting the data in memory area, and code splitting/reconstructing algorithms to handle structured control flow in Wasm. Extensive experiments demonstrate the correctness, effectiveness and efficiency of WASMIXER. Our research has shed light on the promising direction of Wasm binary research, including Wasm code protection, Wasm binary diversification, and the attack-defense arm race of Wasm binaries.

CCS Concepts: • **Security and privacy** → **Security services**.

Additional Key Words and Phrases: WebAssembly, Binary Obfuscation

ACM Reference Format:
Shangtong Cao, Ningyu He, Yao Guo, and Haoyu Wang. 2023. WASMIXER: Binary Obfuscation for WebAssembly. 1, 1 (August 2023), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

WebAssembly (Wasm) is a language-, platform-, and architecture-agnostic binary instruction format, proposed and endorsed by internet giants, including Google and Apple [25]. It aims to offer a both size- and load-time-efficient binary format, which can execute at native speed while running on plenty of platforms, especially web browsers. Therefore, its portability and efficiency make it a strong competitor for JavaScript. For example, lots of computation-intensive jobs are compiled to Wasm and run on browsers already, such as 3D graphic engines [7], cryptocurrency miners [22], and multimedia encoders and decoders [63]. Furthermore, Wasm is moving fast towards a much wider spectrum of domains, e.g., mobile apps [16], IoT [39], blockchain [24], and serverless computing [26].

*Both authors contributed equally to this research.

Authors' addresses: Shangtong Cao, shangtongcao@bupt.edu.cn, Beijing University of Posts and Telecommunications, Beijing, China; Ningyu He, Key Lab on HCST (MOE), Peking University, Beijing, China, ningyu.he@pku.edu.cn; Yao Guo, Key Lab on HCST (MOE), Peking University, Beijing, China, yaoguo@pku.edu.cn; Haoyu Wang, Huazhong University of Science and Technology, Wuhan, China, haoyuwang@hust.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
Manuscript submitted to ACM

arXiv:2308.03123v1 [cs.CR] 6 Aug 2023

https://t.me/learningnets

Though Wasm is a low-level binary format, it can be easily cracked by adversaries. On the one hand, to enable Wasm binaries to be read and edited by humans, Wasm officially provides a human-readable text format [43] (i.e., `wat`), which is an intermediate form designed to be exposed in text editors, browser developer tools, etc. On the other hand, reverse engineering of Wasm is quite easy by taking advantage of emerging binary analyzers for Wasm. For example, *Manticore* [44] is a state-of-the-art static symbolic executor for Wasm. It is able to symbolically execute a function within a Wasm binary and explore multiple paths simultaneously in an abstract way. Through the final results, adversaries can obtain some original semantics of the binary, such as how it will respond against a specific set of inputs. Furthermore, Wasm stores all string literals in a specific area in plain text, which may reflect the code semantics and developers' intentions.

Reverse engineering of Wasm binaries could be used for malicious intentions, e.g., to search for security vulnerabilities in Wasm binaries and based on which to launch attacks [10]. In particular, a recent study [34] suggests that many classic vulnerabilities are completely exposed in Wasm binaries, despite that they are no longer exploitable in native binaries due to common mitigations. Furthermore, many applications on other platforms can be ported to web browser with the help of Wasm, such as Photoshop [4], AutoCAD [1], and some 3D games [3]. Code Obfuscation plays a crucial role in concealing the algorithms and essential logic employed by these software applications. Considering the situation that Wasm binaries are weakly protected, code obfuscation for Wasm becomes an urgent need.

Though it is acceptable to conduct source-level obfuscation before compilation, it is impractical for the case of Wasm. On the one hand, source-level obfuscation techniques are language-specific. They may vary in effectiveness after compiling to Wasm. On the other hand, source-level obfuscation techniques cannot take full advantage of the characteristics of Wasm. For example, *Tigress* [52], a well-known obfuscator for C, only supports the Emscripten compiling chain [23] instead of the Clang version [64], which is often used to compile standalone Wasm binaries. Moreover, it cannot obfuscate the names of imported and exported functions, due to its ineffectiveness on Emscripten's libraries. For this reason, it is necessary to implement a binary-level obfuscator against Wasm.

However, implementing a Wasm-specific obfuscator is challenging, which can be summarized in three-fold. First, obfuscations should bring in as little as overhead in terms of executing time and binary size due to the high efficiency and compact format of Wasm. For example, inserting a function to decrypt all encrypted string literals before executing or at runtime should consider both these two factors. Second, Wasm adopts a highly complicated structured control flow. There are no `goto`-like instructions in Wasm, and such a structure only allows control flow directed in one-way. Implementing control flow obfuscations, such as flattening, while keeping the consistency of semantics is challenging. Third, there are several Wasm-specific static analysis frameworks proposed recently, some of which, especially the symbolic executors, can recover code semantics from obfuscated Wasm binaries. Even introducing opaque predicates, which are effective to evade human inspection, is not enough to bypass those symbolic executors.

This Work. In this paper, to the best of our knowledge, we present the first Wasm-specific obfuscation framework, named WASMixer. It is composed of two main modules, i.e., *data obfuscator* and *code obfuscator*. In general, the data obfuscator is responsible for replacing all function names with random strings, and decrypting all pre-encrypted string literals in Wasm binaries on demand at runtime, which brings in less overhead than one-time decryption at loading. As a complementary, the code obfuscator manipulates instructions and the control flow to resist both human reverse engineering and static analysis, like alias disruption, control flow flattening, and Collatz-based opaque predicates that can resist static analysis. Based on benchmarks consisting of representative Wasm binaries, the evaluation results prove the effectiveness of WASMixer. Specifically, it suggests that obfuscating methods in WASMixer will not bring in any negative impact on the syntactic correctness and semantic consistency. Moreover, WASMixer can also effectively resist

https://t.me/learningnets

manual reverse engineering, hide original intents, and hinder state-of-the-art static analyzers while introducing less than 20% overhead in both terms of executing time and binary size.

Our contribution can be summarized as follows:

- **The first general-purpose Wasm binary obfuscator.** To the best of our knowledge, we have proposed the first language-agnostic binary obfuscation solution for Wasm, named WASMixer, which consists of two modules, specifically against data (string literals and function names) and code (control flow and instructions), respectively.
- **Key techniques for Wasm binary rewriting.** We have proposed a series of key techniques to deal with Wasm binaries, including an on-demand decryption method to minimize the impact brought by decrypting the data in memory area, and code block splitting algorithm and reconstructing interfaces to handle structured control flow in Wasm.
- **Extensive evaluation and useful application scenarios.** We have performed extensive experiments to evaluate the correctness, effectiveness and efficiency of WASMixer, suggesting that it can evade human analysis, state-of-the-art Wasm analyzers and commercial anti-virus engines.

To boost further research on Wasm binaries, we will release WASMixer, along with all the benchmark datasets, to the community.

2 BACKGROUND

In this section, we will briefly illustrate some basic concepts of Wasm as well as code obfuscation.

2.1 WebAssembly (Wasm)

Wasm is an emerging language that can be regarded as the compilation target of multiple mainstream programming languages, e.g., C/C++ [69], Rust [41], and Go [53]. Its advantages mainly reside in its native-like execution speed and compact binary size. We next briefly introduce its features relevant to this paper.

Types & Instructions. There are only four primitive value types defined in Wasm, i.e., `i32`, `i64`, `f32`, and `f64`. The *i* and *f* refer to *integer* and *float*, respectively, while the number corresponds to the length in bits. Wasm has defined over 170 instructions [66] which can consume operands from and produce a return value onto its *operand stack*. For example, `i32.const` pushes its immediate number, expressed as a 32-bit integer, onto the stack.

Data Structure. Wasm has designed a set of simple but effective data structures. Specifically, all data structures in Wasm adopt *key-value* mechanism. For example, each function has its owned *local* structure, local values can be accessed by `local.get i`, where `i` is the index. Data in *global* and *memory* can be accessed and shared by all functions. However, only four primitive types can be stored in the *global* structure, while non-scalar types, e.g., string and array, are stored in the *memory* area. Wasm adopts linear memory, i.e., a string of continuous bytes. A set of instructions are used to load and store data with the memory. For example, `i32.load addr` will take four continuous bytes as a 32-bit integer starting from `addr`.

Wasm Binary. A Wasm binary is composed of *sections*, each of which has its specific functionality. Moreover, a section is regarded as a vector of elements. For example, the *import section* consists of a set of elements, and each element corresponds to an imported function with its name and index. Wasm officially offers a set of tools [55, 56] to support lossless translation between the binary format and a *WebAssembly text format* (wat)[1].

[1]In this paper, all code snippets are wat files converted from Wasm binaries.

https://t.me/learningnets

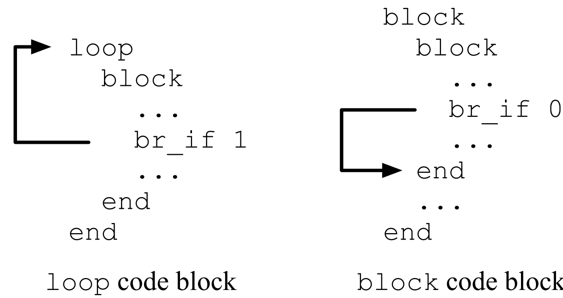

Fig. 1. Two valid control flow jumping in Wasm.

Control Flow. Wasm supports both *direct call* and *indirect call* by `call` and `call_indirect`, respectively. To be specific, a `call` will consume necessary arguments from the stack according to its designated callee, and push the return value (if it has) onto the stack. Instead of directly designating the callee, `call_indirect` takes the top element on the stack as the function index of the callee.

Additionally, Wasm applies a specific *structured control flow*. Specifically, instructions in Wasm are divided into *code blocks*, which can be led by a `block` or `loop` instruction. Code blocks can be nested, but the context of an inner code block is independent to its outer one. Moreover, there are no `goto`-like instructions in Wasm, indicating the control flow cannot be directed to arbitrary instructions. Fig. 1 illustrates the only two allowed control flow jumping rules. As we can see, `br_if` is a conditional jump, i.e., the jump will be performed only if the top element on the stack is not zero. The number following the jump instruction refers to how many layers are intended to jump out (0 is the current one, 1 is its parent, and so on). If the destination code block is led by a `loop`, the control flow will be directed to its heading, or, led by a `block`, the control flow can only go to the tailing `end`.

2.2 Obfuscation

Obfuscation is a widely used software protection method that textually translates programs while preserving semantics to protect the program from being cracked or to prevent sensitive data leakage [38]. Obfuscation methods can be roughly divided into two categories: *data obfuscation* and *code obfuscation*. Specifically, the data obfuscation mainly focuses on readable literals (e.g., function names and string literals) in programs, and translates them into semantic-equivalent but unreadable (or meaningless) ones [19]. The code obfuscation often plays on instructions and the control flow. For example, replacing a single arithmetic instruction with a sequence of bit-shifting instructions, or constructing bogus control flows within a critical function [30].

Obfuscation takes not only humans as its counterparty but also program analysis techniques. Currently, except for the readability, whether an obfuscated program can resist automated program analysis techniques (e.g., symbolic execution) also becomes one of the criteria for evaluating the effectiveness of the obfuscation method [49]. There are generally two kinds of principles [8]. One is to deliberately introduce path explosion issue by constructing complicated control flow. To this end, symbolic executors have to place significant time and resources on meaningless paths constructed by the obfuscator. The second is to exploit the backend SMT solver, like introducing non-linear formula, to make it hard for symbolic executors to determine whether a path is feasible or not.

https://t.me/learningnets

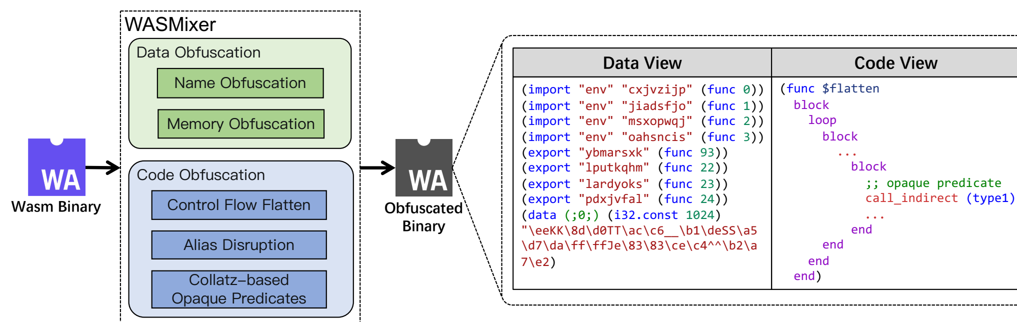

Fig. 2. The architecture and workflow of WASMixer.

3 WASMIXER

In this section, we first overview the architecture of WASMixer, and then illustrate the necessity of implementing such a binary-level obfuscator. Last, we also detail the challenges in designing and implementing WASMixer.

3.1 Overview

WASMixer takes a Wasm binary as input and generates an obfuscated binary based on the specified options, as shown in Fig. 2. As we can see, WASMixer is mainly composed of two components: *data obfuscator* and *code obfuscator*. Data obfuscator can perform name and memory obfuscation. The purpose of name obfuscation is to replace names of functions, import and export items, and other identifiers with random strings, while memory obfuscation encrypts the initialized linear memory in Wasm binary and decrypts it at runtime. Moreover, WASMixer also proposes three code obfuscation methods, including flattening obfuscation for control flow, alias disruption for `call` instructions, and Collatz-based opaque predicates for resisting static analysis, e.g., symbolic execution. These obfuscation methods can effectively resist both human reverse engineering and program analysis. The implementation of WASMixer is detailed in §4.

3.2 Why Binary Obfuscator?

Compared with source code and compiler IR level obfuscators, the biggest advantage of obfuscating Wasm binaries is the *compatibility*. Specifically, on the one hand, source code level obfuscation inherently has limitations when source code is unavailable, especially when dealing with a large number of legacy binaries. In addition, this lack of direct connection to the source code isolates the source semantics, making it more challenging for attackers to analyze. For instance, the relationship between Wasm instructions can be converted to an Abstract Syntax Tree (AST) through stack operations, where the ASTs hold source code semantics. Binary obfuscation can construct uncommon stack instruction snippets that obfuscate the semantics of the generated AST. On the other hand, obfuscating methods for diverse source programming languages are different from each other, resulting in significant differences upon obfuscating effectiveness on compiled Wasm binaries. In general, obfuscators for compiled languages have more methods to protect programs, while compression and encryption are more commonly used obfuscation methods for scripted languages. Binary obfuscation is not limited by language type and can provide obfuscations at the same level across all languages. Moreover, source code level obfuscation inherently has limitations when source code is unavailable, especially when

https://t.me/learningnets

Table 1. Comparison between Tigress and WASMIXER.

Functions	Tigress	WASMIXER
support multi-languages	×	✓
unavailable source code	×	✓
supported compilers	emscripten	all
obfuscating import & export	×	✓
obfuscating debug information	×	✓
against symbolic execution	×	✓

dealing with a large number of legacy binaries. Moreover, as mentioned in §4.2.2, the Wasm language has its own unique and effective binary obfuscation methods that are not covered by previous obfuscators.

Take one of the most popular obfuscation tools for C programming language, *Tigress* [52], as an example. For clarification, we summarize the pros and cons of Tigress and WASMIXER in Table 1. We can see that in addition to compatibility issues, i.e., multi-language supporting and unavailability of source code, we also find that Tigress can only support the Emscripten [23] compiler, without supporting compilers like clang [64], which is often used for compiling standalone Wasm binaries. Additionally, Tigress is unable to obfuscate the library provided by Emscripten, which indicates that both the import and export items and the debug information cannot be replaced by meaningless strings. Last, although Tigress provides complex obfuscation methods such as virtualization, they can still not resist symbolic execution, like KLEE [32]. Consequently, our binary-level obfuscator can fill the gap and overcome the inherent shortcomings of source code level obfuscators.

3.3 Challenges

To the best of our knowledge, we have implemented the first binary obfuscator against Wasm binaries. Although there are several existing efforts [6, 37, 54, 58] on Wasm binary analysis, they are limited to instruction-level rewriting, e.g., Wasabi [54] and Wasm-mutate [6], or simple control flow translation, e.g., Fuzzm [37]. Especially, in implementing WASMIXER, we face mainly three challenges.

C1: Low-overhead & General Data Obfuscation. Except for the implementation and declaration of functions, a Wasm binary contains lots of data, e.g., initial memory data and debugging information. Effectively obfuscating these data without affecting semantic consistency is challenging. For instance, inserting a decryption function between the initiation and the execution of a Wasm binary will significantly increase the response time, which is harmful to users' experience. Or, the decryption can be performed on demand during executing `load` instructions, e.g., by substituting or hooking these instructions. However, there are more than 20 `load`- and `store`-related instructions in Wasm for different bit-length. Implementing and inserting multiple hooking functions in a single Wasm binary is impractical and inefficient. Additionally, Wasm does not mandate the format of various types (e.g., variable types and function names) of debugging information in its *custom section*, which may vary according to different source languages.

Our Solution: To minimize the impact brought by decrypting the data in memory area, we propose an on-demand decryption method. For different memory loading and storing instructions, the method can dynamically obtain the destination address, and determine how many bits should be loaded or stored. Then, a runtime decryption or encryption will be conducted accordingly. As for obfuscating the function names in debugging information, we also design a parsing

https://t.me/learningnets

method that can precisely extract function names with variable length, and replace them with random characters without breaking the original structure.

C2: Structured Control Flow. As we illustrated in §2.1, Wasm adopts the structured control flow, where the control flow can only be jumped sequentially, i.e., from an inner code block to outer ones. Moreover, destinations of jump instructions cannot be designated arbitrarily. They have to be the head (led by `loop`) or the tail (led by `block`) of the targeted code block. Without the help of `goto`-like instructions, it is challenging to modify the control flow in Wasm to perform obfuscation. Moreover, code blocks are independent to each other. In other words, variables cannot be shared across code blocks. To this end, some obfuscating methods that require splitting code blocks, e.g., control flow flattening [33], cannot be conducted easily while keeping the semantics intact.

Our Solution: An important prerequisite for accomplishing code obfuscation is to ensure semantic consistency. With this mandatory requirement in mind, we first propose a code block splitting algorithm, which can split a given code block into several ones ensuring the stack balance of new blocks and variable sharing across blocks meanwhile. Then, we propose another algorithm, named code block rearranging algorithm, which can rearrange the split code blocks in a flattened manner. To this end, the control flow of any given Wasm code blocks can be flattened, which achieves the goal of obfuscation.

C3: Effective Obfuscation against Static Analysis. There are two counterparties for program obfuscation, i.e., human reverse engineering and static analyzers. Increasing unreadability, e.g., memory encryption or function names obfuscation, on Wasm binaries may be effective for countering human, but not for static analyzers, e.g., symbolic executor. It is because symbolic executors can recover some semantic information to some extent. For example, the result of an opaque predicate cannot be easily obtained by a hacker, but it is a piece of cake for symbolic executor. Therefore, we should deliberately invalidate symbolic execution on Wasm level.

Our Solution: In order to solve this problem, we decided to take advantage of the inherent limitation of symbolic execution, i.e., path explosion. We will introduce path explosion by two methods. On the one hand, we replace all static calls with indirect calls, which has to force symbolic executors to try all possible callees because the callee will only be determined at runtime. On the other hand, we introduce an opaque predicate, in which it has an unbounded loop. And we deliberately pass an undetermined value (symbol) as the input of this opaque predicate. To this end, symbolic executors have to maintain an exponentially growing number of paths at every calling point to this opaque predicate.

4 APPROACH

In this section, we will depict the technical details of data obfuscator and code obfuscator in WASMixer.

4.1 Data Obfuscator

As we mentioned in §2.2, data obfuscation is used to increase the difficulty of reading and understanding the data by either human or automated analyzers. For Wasm binary, the data can be roughly divided into the linear memory and the debugging information.

4.1.1 Memory Obfuscation. In a Wasm binary, memory data is stored in plaintext. It mainly composed of constant strings in the source program, like string templates in `printf` and `scanf` [29]. To some extent, such a way of storing data in plaintext will inevitably reveal the code semantics and developers' intentions, facilitating the cracking of Wasm binaries. It is critical, therefore, to obfuscate the memory data declared initially or even generated dynamically.

https://t.me/learningnets

Algorithm 1: On-demand & Runtime Memory Loading and Storing Algorithm.

```
   Input: base - base address of target,
   offset - offset address of target,
   signed - padding as signed or not,
   len - the length of loaded data in bits,
   type - the target type
   Output: data - loaded data
 1 Procedure dec_load(base, offset, signed, len, type)
 2 |  address ← base + offset
 3 |  data ← loadMemory(address, len)
 4 |  key ← getKey()
 5 |  data ← xor(data, key)                    {Decryption}
 6 |  if signed then
 7 |  |  data ← signedExtend(data, type)
 8 |  else
 9 |  |  data ← unsignedExtend(data, type)
10 |  end
11 |  return data
12 end
13 Procedure enc_store(base, offset, signed, type)
14 |  key ← getKey()
15 |  data ← xor(data, key)                    {Encryption}
16 |  data ← trunc(data, type)
17 |  address ← base + offset
18 |  storeMemory(address, type, data)
19 end
```

To decrypt the encrypted memory data, two strategies can be applied: *decrypting at the entry* or *decrypting on-demand*. Specifically, as for the former method, a decryption function will be inserted as a wrapper at the entry function. Once the binary is initiated, its memory data will be decrypted all at once. However, it will introduce a huge overhead. On the one hand, not all data will be retrieved during an actual execution. Such decrypting all data will consume unnecessary time and resources. On the other hand, decrypting with the binary initiation will increase its response time, decreasing the overall experience. Note that, the data generated during the runtime is not encrypted. Therefore, if the runtime memory can be read through certain vulnerabilities (e.g., out-of-bound reading [34]), it can also lead to data leakage.

To resolve the **C1** (see §3.3), we propose an algorithm that can load (store) data from (to) memory on-demand with a runtime encryption to guarantee the data confidentiality, as shown in Algorithm 1. As it shows, we first replace all `load` and `store` instructions as `call dec_load` and `call enc_store`, respectively. Let us take the `dec_load` as an example. There are 14 `load` instructions defined in the Wasm specification. The instruction `i32.load8_u` will load 1 byte from the target address, extend it as a 4-byte unsigned integer, and push it onto the stack. Therefore, in the implementation of `dec_load`, we first calculate the target address by `base` and `offset`, both of which are pushed onto the stack already. Then, after loading the data according to `len` (L3), the key will be loaded and XOR-ed with the loaded data (L5). Because the XOR operation is symmetric, the key is generated randomly and kept in both `dec_load` and `enc_store`. At last, the loaded data will be extended according to the target type. The process plays similarly in `enc_store`. Note that, at L15,

https://t.me/learningnets

we also conduct the XOR operation on the key with the to-be-stored data, which keeps the confidentiality of runtime data.

4.1.2 *Name Obfuscation.*

Name obfuscation aims to rename identifiers, e.g., variables and functions, to avoid malicious cracking and prevent data leakage [70]. In Wasm, there are two types of identifiers should be protected, i.e., readable function names in debugging information, and function identifiers in import/export sections.

Debugging information in Wasm is stored in the custom section, which is generated by compilers directly. It consists of some sensitive metadata of the binary, e.g., compiler version, function names, variable types, and even the path of its source code [35]. The format of function names is specified as follows:

$$\texttt{name}||\texttt{0x1}||len_{sec}||\overbrace{num_{name}||\underbrace{(idx_i||len_{name_i}||name_i)}_{\text{repeat } num_{name} \text{ times}}}^{\text{length in } len_{sec} \text{ bytes}}$$

, where the leading `name` and `0x1` are string literals, and the length of each function name is declared by its corresponding len_{name_i} [65]. Therefore, to hinder reverse engineering, we keep everything intact except for $name_i$. We replace them with random strings with the same length. In this way, execution logic and call relationships for all functions are identical to the original ones, but function names have been changed to meaningless strings for human.

Wasm allows the import functions from or exports its implementation of functions to its environment. These two parts are declared in the import and export section, respectively. Take the export section as an example. For each declaration in the export section, it is composed of $name_{internal}$ and $name_{export}$, i.e., exporting the internal function, dubbed $name_{internal}$, as $name_{export}$. To this end, the outer environment (like web browsers) can call the function $name_{export}$ through JavaScript [42]. However, the plaintext of $name_{export}$ can somewhat reflect its intention, which may be used as tokens by detectors [29]. We offer an option to substitute random characters for $name_{export}$. It is worth noting that a small side effect of this obfuscating method is that it requires modifying outer modules to call the random strings instead of $name_{export}$. Thus, we offer an auxiliary tool to assist developers to update the call to $name_{export}$ as the random string in its outer environment.

4.2 Code Obfuscator

As we mentioned in §2.2, code obfuscation plays on instructions and control flow. In this section, we will introduce how we deal with Wasm-specific instructions and its structured control flow.

4.2.1 *Control Flow Obfuscation.*

Control flow obfuscation [11] is a set of mainstream and effective obfuscating methods that complicate program's control flow to make it unreadable and hard to analyze. Typically, it involves inserting new control flows to confuse analyzers, or modifying and complicating existing control flows. Due to the complicated structured control flow adopted by Wasm, it is challenging to achieve that. Hence, we abstract performing control flow obfuscation in two major steps, i.e., *code block splitting* and *code block rearranging*. Except for detailing these two steps, we also demonstrate how to take advantage of them to conduct *control flow flattening* [33].

Stage I: Code Block Splitting. The structured control flow applied by Wasm makes it challenging for performing code block splitting as we mentioned in **C2**. Specifically, in Wasm, code blocks are independent to each other. In other words, variables on stack that can be accessed by a code block cannot be retrieved after splitting it into several ones. In summary, splitting instructions into several code blocks requires support specifically against Wasm.

https://t.me/learningnets

Algorithm 2: Code Block Splitting Algorithm.

```
   Input: cb - the target code block,  num - the number of code blocks intended to split
   Output: cbs - generated code blocks
 1 funcIdx ← extractFuncId(cb)
 2 maxLen ← getMaxStackLength(cb)
 3 appendFuncLocal(funcId, maxLen)
 4 cbs ← listDivide(cb, num)
 5 foreach cb in cbs do
 6   | stackPost ← getPostStack(cb)
 7   | stackPre ← getPreStack(cb)
 8   | foreach type in stackPost do
 9   |   | modifyLocalType(funcId, type)
10   |   | insertInstruction(cb, type, local.set)                {Store stack}
11   | end
12   | foreach type in stackPre do
13   |   | modifyLocalType(funcId, type)
14   |   | insertInstruction(cb, type, local.get)                {Restore stack}
15   | end
16 end
17 return cbs
```

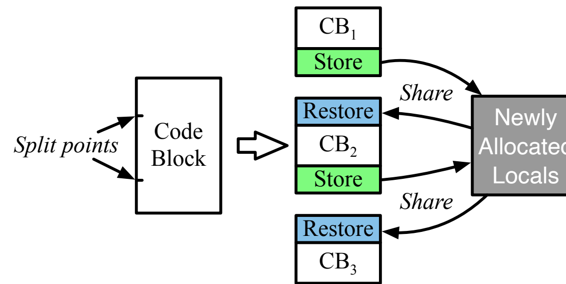

Fig. 3. Split a code block into separate ones, among which the arguments are shared through newly allocated locals.

We propose a code block splitting algorithm, which is shown in Algorithm 2. It takes a code block and a number that refers to how many blocks are intended to split as inputs. The algorithm firstly extracts which function the target block locates in (L1), and calculates the maximum size of used stack slots according to its contained instructions (L2). To this end, it can determine how many locals (as temporary variable sharing area between newly split code blocks) should be newly allocated in the function (L3). After splitting the code block into several ones, storing and restoring stack elements should be conducted after and before each of them, respectively. As illustrated in Fig 3, each newly split code block is ended with a list of instructions to store necessary variables, and started with instructions that restore them. Take the stack storing as an example, the algorithm will traverse a newly generated code block to determine the number and the types of elements remained, which are maintained in *stackPost* at L6. According to the types of remained elements, L9 modifies the type of the corresponding local. An extra `local.set` will be appended to the code block, thus remained elements can be temporarily stored in local variables, which can be accessed by the restoring

https://t.me/learningnets

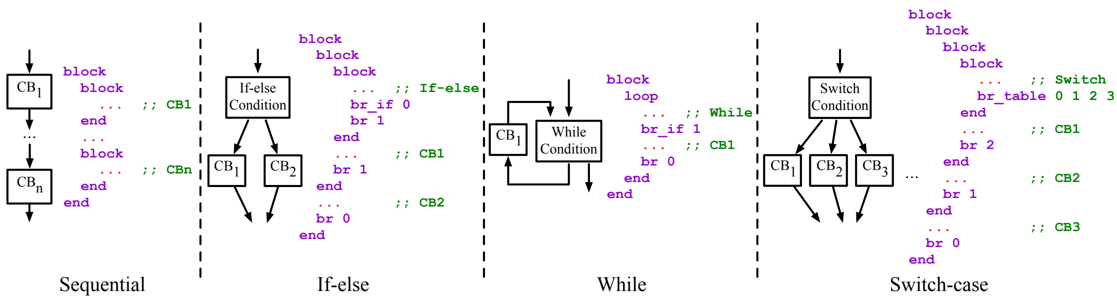

Fig. 4. Four basic relationships among code blocks.

process of the next code block. Consequently, a given code block is split into several tiny code blocks, among which variables could be shared between newly allocated locals.

Stage II: Code Block Rearranging. The code block splitting algorithm can only split a code block into several ones, where they still keep a sequential relationship. As we mentioned in **C2**, there is no `goto`-like instructions that can arbitrarily guide the control flow. The instructions `br` and `br_if` can only jump to designated code blocks' beginning or tailing (see §2.1). It is challenging and requisite to implement a set of interfaces that allow users to assemble given code blocks according to various relationships on demand without considering the correctness of syntax. To this end, we have summarized four basic relationships among code blocks, i.e., *sequential*, *if-else*, *while*, and *switch-case*, as shown in Fig. 4.

As we can see, the basic elements to compose these four relationships are code blocks and a condition (except for the sequential one). In other words, once given enough elements, a structure following the designated relationship will be assembled. For example, the interface of if-else relationship is:

```
assembleIfElse(cond, cb1, cb2)
```

, where `cond` is a condition statement, and `cb1` and `cb2` are code blocks either constructed by users or generated by the code block splitting algorithm. In either case, users can complete the rearrangement without concerning details of syntactic correctness.

Case Study: Control Flow Flattening. Based on the code block splitting algorithm and interfaces exposed by code block rearranging, we can easily implement the control flow flattening obfuscation. In general, control flow flattening is to split a series of instructions into code blocks, which are cases for a switch-case statement. Such a switch-case will be wrapped by a loop to execute recursively till all cases are executed once in the original order. Thus, we have implemented a flattening obfuscation, which is shown in Algorithm 3. Suppose it takes a series of split code blocks as input. Specifically, we first record the original order of the given code blocks and shuffle them for enhancing the obfuscation effect (L1 and L2). To pass the index of the to be executed code block, we have to allocate another local variable, which is done by L3 and L4. Then, for each code block, we append a list of instructions (after the variable storing, see Fig. 3) that can jump back to the switch condition, and pass an index that refers to the successive block through an unconditional jump (L5 to L10). Afterward, we construct a switch condition (achieved by `br_table`), and assemble it with the code blocks mentioned above into a switch-case statement (L11 and L12). Moreover, we should

https://t.me/learningnets

Algorithm 3: Control Flow Flattening.

```
   Input: cbs - a list of code blocks
   Output: flattenCB - a code block composed of given code blocks with flattened relationship
 1 cbsOrder ← recordOrder(cbs)
 2 shuffle(cbs)
 3 funcIdx ← extractFuncId(cbs)
 4 jumpFlagIdx ← appendFuncLocal(funcIdx)
 5 foreach cb in cbs do
 6 |   postCB ← findSuccCB(cbsOrder, cb)
 7 |   appendLocalAssign(cb, jumpFlagIdx, postCB)
 8 |   nestedLevel ← getNestedLevel(cbsOrder, cb)
 9 |   appendJumpInstr(cb, nestedLevel)
10 end
11 switchCond ← createSwitchCond(jumpFlagIdx, cbsOrder)
12 switchCaseCB ← assembleSwitchCase(switchCond, cbs)
13 whileCond ← appendExit(switchCaseCB, jumpFlagIdx)
14 flattenCB ← assembleWhile(whileCond)
15 return flattenCB
```

further wrap the switch-case into a while statement to achieve iterating on each case (L13 and L14). To this end, each code block will be executed once in the original order.

Fig. 5 shows a concrete example of the structure of flattened code blocks. As we can see, three code blocks are given and flattened in an order different from the original one. For each block, it begins with variable restoring instructions, and ends with variable storing instructions as well as instructions that designate its successive code block. In this example, the original sequential order is shuffled, i.e., the first case (L14 to L18) will be executed at the second place, which increases the difficulty for human reverse engineering to some extent. Note that, according to the *while* structure defined in Fig. 4, both the switch-case statement and the exit instruction (L31) are the condition of the while statement, while its body is an empty code block, which should be inserted between L32 and L33. Our rearranging interfaces regard such empty code blocks as valid situations. Consequently, we can conclude that by combining the code block splitting algorithm and the interfaces of code block rearranging, we can achieve code block flattening in Wasm.

4.2.2 Alias Disruption Obfuscation. As we mentioned in §2.1, in Wasm, function invocation can be achieved using either `call` or `call_indirect`, while the former one, static call, is widely used and can be easily analyzed. Replacing the former with the latter one can achieve alias disruption obfuscation to some extent.

Specifically, a static call will designate its callee directly, like `call $foo`. However, the indirect call, named `call_indirect`, calls a function according to the top element of the stack at runtime. It is designed for function pointers and polymorphism in high-level languages. As explained in Daniel [36], `call_indirect` is hard to analyze for determining the index value of an indirect function is challenging. Since the type of index value is `I32`, which has no unique characteristic to assist analysis. Furthermore, such a runtime-determined callee will also hinder the static analysis, especially for enumerating possible callees [43]. In a Wasm binary, there is a special section, named *elem section*, where it declares all possible callees of `call_indirect` by a list of function references at a specific offset. To perform inter-procedural analysis, the static analyzer has to enumerate all possible callees declared in the *elem section*.

https://t.me/learningnets

```
1   (func $flatten
2     (local $jumpFlag i32)
3     (local i32 i64 f32 f64 ...) ;; store operand stack
4     ...  ;; set $jumpFlag = 0
5     block
6       loop
7         block
8           block
9             block
10              block
11               local.get $jumpFlag
12               br_table 0 1 2 3
13   ①②③       end
14              ... ;; restore stack instructions
15              ... ;; CB2                              ②
16              ... ;; store stack instructions
17        ②     local.set $jumpFlag
18              br 2
19            end
20            ...   ;; CB1                              ①
21            ...   ;; store stack instructions
22        ①   local.set $jumpFlag
23            br 1
24          end
25          ...     ;; restore stack instructions
26          ...     ;; CB3                              ③
27          ...     ;; store stack instructions
28        ③ local.set $jumpFlag
29          br 0
30        end
31        ...       ;; judging whether to exit
32        br_if 1
33        br 0
34      end           ④
35    end)
```

Fig. 5. A code snippet to illustrate a flattened control flow.

To achieve such an alias disruption obfuscation, many steps are required. First, we need to obtain the indices of callees of all `call` instructions, and append them into the *elem section*. Then, before each to-be-replaced `call` instructions, we insert an `i32.const` instruction with its callee's index. Finally, we will examine the type of the callee, and pass the type index as the argument of the replaced `call_indirect`[2]. For example, a `call $foo` can be replaced by:

```
i32.const 7
call_indirect (type 2)
```

, where we assume the function `$foo` is the 7th function, and its type is declared as the 2nd one.

To further improve the obfuscating effect, we take advantage of the opaque predicate [68] instead of an `i32.const` instruction. Specifically, an opaque predicate refers to an expression whose result is constant regardless of the input. Typically, the returned value of an opaque predicate is only known to its obfuscator, and is difficult for analyzers to infer. A simple opaque predicate, e.g., `x(x-1)%2`, will always return `0` regardless of the value of `x`. We can replace some of `i32.const` before `call_indirect` instructions into opaque predicates to improve the effectiveness of the alias disruption obfuscation.

[2]Pass the callee's type index is mandatory for `call_indirect` to guarantee the stack balance.

https://t.me/learningnets

4.2.3 Collatz-based Opaque Predicate Obfuscation. Currently, in Wasm, several static symbolic execution tools have been proposed, like manticore [44] and WANA [57]. Theoretically, with the help of symbolic executors, attackers can retain the semantics of Wasm binaries easily, which may invalidate the obfuscation. To this end, we have implemented a collatz-based opaque predicate obfuscating method in Wasm to evade the analysis of such tools.

One of the mainstream methods to counter symbolic execution is deliberately introducing path explosion. For example, introducing an unbounded loop that has no effect on original semantics. We introduced *Collatz conjecture* [2], a famous unsolved problem in mathematics also known as 3n + 1 conjecture, to achieve this goal. The conjecture asserts whether repeating two simple arithmetic operations will eventually transform every positive integer into 1. Specifically, the two operations are:

$$collatz(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \\ 3n+1 & \text{if } n \equiv 1 \end{cases} \pmod 2$$

, where the conjecture takes a positive integer and checks whether it is odd or even. Most importantly, there is no general term formula for Collatz conjecture. In other words, we cannot determine how many rounds are required for the final convergence on 1.

Note that, although our fellow researchers have applied the Collatz conjecture to code obfuscation in previous work [59], we are the first to implement it in Wasm. During symbolically executing the function *collatz*, if the input is or contains any symbolic values, it will conduct path forking on whether the given n is odd or even. After several rounds, the number of paths will increase exponentially, which requires lots of computing- and space-resources. Therefore, we use it as an opaque predicate to construct the *jumpFlag* in both control flow flattening (see §4.2.1) and alias disruption obfuscation (see §4.2.2). Take constructing `jumpFlag` in Fig. 5 as an example, and we assume the value of `jumpFlag` to be assigned is 3. To this end, the assignment expression for the `jumpFlag` is:

```
1 a = m * x + c
2 while (collatz(a) > 1):
3   a = collatz(a)
4 jumpFlag = collatz(a) + 2
```

, where x is a symbol, which generally comes from one of the parameters of the function, and m and c are all random integers. To this end, the final value of `jumpFlag` will always be 3 regardless of the values of x, m and c. However, in this way, the function `collatz` will take a monomial as input, which can be easily solved if the context of symbolic variable is already constrained. For example, if the constraint of x is already limited in $[0, 3]$, only four paths need to be searched. In order to increase the search space and thus reduce the analysis efficiency, we can replace the a at L1 with a polynomial, like:

$$a = m * x + n * y + c$$

, where y is also a parameter in the function. Even if y is limited in a constraint, e.g., $[5, 8]$, the number of paths is the Cartesian product of possibilities of x and y. Thus, there are 16 (4*4) paths are required to be searched. If it is necessary, more terms can be inserted into the polynomial. Consequently, symbolically executing this piece of code is extremely inefficient due to the exponentially increased number of paths imported by the Collatz conjecture.

https://t.me/learningnets

5 IMPLEMENTATION & EVALUATION

5.1 Implementation

We have implemented WASMIXER with over 5.9K LOC of Python3 code from scratch. To avoid reinventing the wheel, we implement the Wasm binary decoder based on wasm-python-book[3], a simple but effective Wasm runtime. We have packaged WASMIXER into a standard Python library. Based on the exposed obfuscating methods, security developers can conveniently apply them to specified Wasm binaries.

5.2 Research Questions & Experimental Setup

Our evaluation is driven by the following three research questions:

RQ1 Can those implemented obfuscating methods in WASMIXER keep semantic consistency?

RQ2 How effective are the provided obfuscation methods against human reverse engineering, malware detectors and static analyzers?

RQ3 How much overhead are brought by WASMIXER in terms of runtime and code size?

To answer the above RQs, we have collected a sophisticated set of Wasm binaries, which are shown in Table 2. Specifically, to answer **RQ1**, we need some binaries that have explicit and determined input and output. Basic Algorithms [45] ($\mathbf{D}_1$) is composed of some commonly used algorithms, like string concatenation and quick sort on an array, compiled from C to Wasm. Moreover, we also collected two real-world applications (base64 and md5) found on wapm [60] ($\mathbf{D}_2$), a well-known and mainstream Wasm package manager.

To answer **RQ2**, we have collected three datasets ($\mathbf{D}_3$ to $\mathbf{D}_5$) and three representative analyzers against Wasm binaries (VirusTotal, manticore, and wasp). To be specific, we chose VirusTotal [5], the most authoritative and widely-used malware analysis service currently, which is supported by more than 60 security vendors, i.e., each target will be scanned by more than 60 detectors. Based on SEISMIC [58] and MinerRay [47], we have collected 18 Wasm binaries that are labeled as malicious cryptominers as the ground truth, which dataset is denoted as $\mathbf{D}_3$. As for the static analyzers, we chose two state-of-the-art symbolic executors, i.e., *manticore* [44] and *wasp* [40]. Specifically, manticore is a commercial and actively maintained native symbolic executor for Wasm binaries. It provides several features, e.g., fine-grained control of state exploration via event callbacks. Different from manticore, wasp is a concolic executor. In other words, it dynamically executes the given Wasm binary with random inputs and records all path conditions. According to the solutions returned by its backend SMT-solver, wasp can mutate the seed and reach a high coverage. To evaluate the effectiveness of our proposed obfuscating methods against symbolic execution, we enforced both manticore and wasp to execute cases from the btree-traverse [62] ($\mathbf{D}_5$). These cases perform a process of initiating, inserting, traversing, and deleting on btrees with different orders. Moreover, Gillian-Collections-C [21] ($\mathbf{D}_4$) is a test suite for testing the Collections-C [20], a well-known library for common data structures implemented in C. However, because wasp only supports a subset of Wasm instructions, only manticore is set to analyze $\mathbf{D}_4$. Last, we have evaluated its ability to resist human reverse engineering on all $\mathbf{D}_3$, $\mathbf{D}_4$ and $\mathbf{D}_5$, using a number of metrics.

Last, to answer **RQ3**, we measured the imported overhead on all datasets except for $\mathbf{D}_3$. This is because these 18 cryptominers all run on browsers, requiring lots of dependency modules and interactions with the corresponding JavaScripts. Therefore, we only measure the overhead on those standalone cases.

All experiments were performed on a server running Ubuntu 22.04 with a 64-core AMD EPYC 7713 CPU and 256GB RAM.

[3]https://github.com/Relph1119/wasm-python-book (commit: 0x872bc8f)

https://t.me/learningnets

Table 2. Datasets used by different RQs.

	Datasets	#Wasm Binaries
RQ1	Basic Algorithms [45] ($\mathbf{D}_1$) wapm programs [60] ($\mathbf{D}_2$)	18 2
RQ2 (VirusTotal)	cryptomining programs [47, 58] ($\mathbf{D}_3$)	18
RQ2 (manticore)	Gillian-Collecitons-C [21] ($\mathbf{D}_4$) btree-traverse [62] ($\mathbf{D}_5$)	159 32
RQ2 (wasp)	btree-traverse [62] ($\mathbf{D}_5$)	32
RQ3	$\mathbf{D}_1$, $\mathbf{D}_2$, $\mathbf{D}_4$, $\mathbf{D}_5$	243

5.3 RQ1: Semantic Consistency

As an obfuscator, maintaining semantic consistency is a necessary condition that must be met. To this end, we must evaluate if a Wasm binary can keep identical semantics before and after the obfuscation of WASMixer. In total, we choose 20 cases from $\mathbf{D}_1$ and $\mathbf{D}_2$ (see Table 2) as candidates. Specifically, all these 20 cases can generate explicit and determined output when given a set of inputs. For each of them, we generate 14 different obfuscated versions according to different obfuscating options. The two options are applying memory and name obfuscation (see §4.1), respectively, while the other 12 options are divided into two groups, the only difference between these two groups is enabling the Collatz-based opaque predicate obfuscation (see §4.2.3) or not. Further, each group consists of 6 cases, corresponding to the number of split code blocks as 5, 10, and 20 in control flow flattening obfuscation (see §4.2.1) and performing alias disruption obfuscation (see §4.2.2) on 25%, 50%, and 100% candidate instructions.

For each candidate case, along with its 14 mutated versions, we construct sets of inputs to cover as many program paths as possible. For example, for an argument type as `unsigned char`, we will randomly choose its value ranging from 0 to 255. For a string, we will construct a string with random printable characters whose length is from 0 to 10. Moreover, as each Wasm binary will be statically verified before executing [67], we also asked wasm-validate[4] to examine the syntactic correctness of these 300 cases.

Results show that with different obfuscating options, all 300 obfuscated Wasm binaries can still pass the verification of wasm-validate, which indicates that WASMixer will not corrupt the syntax of given Wasm binaries. Moreover, with identical sets of input, obfuscated cases can generate the same output as the original one. We can conclude that WASMixer will not bring in any negative impact towards syntax and semantics during obfuscating.

RQ-1 Answer

By combining and applying 5 proposed obfuscating methods on 20 Wasm binaries with determined output, results show that WASMixer will not bring in any negative impact on the correctness of syntax and semantic consistency.

5.4 RQ2: Effectiveness

The effectiveness of obfuscation can be evaluated from three perspectives. First, obfuscated Wasm binaries should be unreadable for malicious users who typically can arbitrarily access them. Second, obfuscated Wasm binaries can

[4]An official tool to verify the validity of Wasm binaries.

https://t.me/learningnets

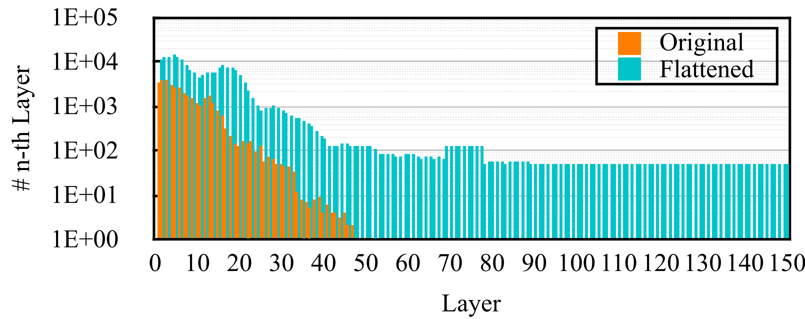

Fig. 6. The distribution of n-th nesting layers before and after control flow flattening for Wasm binaries in $\mathbf{D}_1$ to $\mathbf{D}_5$.

Table 3. The number of `call_indirect` and the length of element section for Wasm binaries in $\mathbf{D}_4$ and $\mathbf{D}_5$ before and after `call` to `call_indirect` obfuscation.

	#call_indirect		**length of the elem section**	
	Original	Obfuscated (times)	Original	Obfuscated (times)
$\mathbf{D}_1$	234	3,180 (13.6x)	72	1,110 (15.4x)
$\mathbf{D}_2$	188	5,423 (28.8x)	253	963 (3.8x)
$\mathbf{D}_3$	92	2,153 (23.4x)	282	1,446 (5.1x)
$\mathbf{D}_4$	3,554	32,998 (9.3x)	1,159	11,868 (10.2x)
$\mathbf{D}_5$	0	606 (N/A)	0	198 (N/A)
Total	**4,069**	**44,360 (10.9x)**	**1,766**	**15,585 (8.8x)**

effectively hide their original intents. Third, obfuscated Wasm binaries should be hard for analyzing by static analyzers. For example, for a static symbolic executor, the analyzing time for an obfuscated Wasm binary should be unacceptable. We evaluate the effectiveness of WASMixer in the following.

5.4.1 Against Manual Reverse Engineering. As we mentioned in §2.1, for a valid Wasm binary, users can convert it to a readable text file, i.e., wat file, through an official toolkit. Four obfuscating methods can significantly increase the unreadability of a wat file, i.e., *name obfuscation*, *memory obfuscation*, *control flow flattening*, and *alias disruption*. Because it is obvious that name and memory obfuscation can resist manual reverse engineering effectively, we only evaluate the other two methods in this section. Specifically, the flattening will deliberately split a code block and insert lots of jump instructions. Such a frequent and nested control flow jumping between code blocks will definitely confuse malicious users. Moreover, by applying alias disruption, once we convert `call` to `call_indirect`, its operand, i.e., the callee, will not be statically designated. In other words, malicious users have to consider all functions listed in the elem section as possible callees (see §4.2.2).

Therefore, we have applied both methods on all cases from $\mathbf{D}_1$ to $\mathbf{D}_5$. Figure 6 illustrates the distribution of n-th nesting layers after the control flow flattening is conducted. As we can see, there is an explicit shift towards the right after performing flattening, which indicates that the number of layers with high orders increases. The results show that, for n-th nesting layers ($n > 5$), the numbers for cases in $\mathbf{D}_1$ to $\mathbf{D}_5$ increase by 7.14 times. Moreover, before flattening, the highest n for is 72 (occurs in a case in D_3), while it turns to 489 after flattening. It can be proven that *control flow*

https://t.me/learningnets

Table 4. The results of VirusTotal on identifying cryptomining Wasm binaries with different obfuscation options. OB_1 to OB_5 refer to name obfuscation, memory obfuscation, control flow flattening, alias disruption, and the combination of name and memory obfuscation, respectively.

		Name Obfuscation			Code Obfuscation	
ID	**original**	**OB_1**	**OB_2**	**OB_5**	**OB_3**	**OB_4**
dd8aabc0	32	15 (53%)	7 (78%)	0 (100%)	32 (41%)	19 (0%)
8e2fc626	19	2 (89%)	11 (42%)	0 (100%)	14 (26%)	14 (26%)
eeeaaf2e	17	3 (82%)	11 (35%)	0 (100%)	13 (12%)	15 (24%)
74a4ed58	14	3 (79%)	2 (86%)	0 (100%)	7 (50%)	7 (50%)
9cda640f	7	0 (100%)	1 (86%)	0 (100%)	4 (14%)	6 (43%)
b170c14e	7	2 (71%)	3 (57%)	0 (100%)	5 (14%)	6 (29%)
a2b93b31	6	2 (67%)	0 (100%)	0 (100%)	5 (17%)	5 (17%)
73c6c519	5	2 (60%)	0 (100%)	0 (100%)	3 (0%)	5 (40%)
50ca16c9	4	2 (50%)	1 (75%)	0 (100%)	4 (0%)	4 (0%)
e742774b	3	2 (33%)	0 (100%)	0 (100%)	2 (33%)	2 (33%)
37ff6a30	3	0 (100%)	0 (100%)	0 (100%)	3 (0%)	3 (0%)
41087519	2	2 (0%)	0 (100%)	0 (100%)	2 (0%)	2 (0%)
1f925c46	2	2 (0%)	0 (100%)	0 (100%)	2 (0%)	2 (0%)
5bec2bbb	2	2 (0%)	0 (100%)	0 (100%)	2 (0%)	2 (0%)
f879f0ce	2	2 (0%)	0 (100%)	0 (100%)	2 (0%)	2 (0%)
64709eac	2	2 (0%)	0 (100%)	0 (100%)	2 (0%)	2 (0%)
c0d72dbb	2	2 (0%)	0 (100%)	0 (100%)	1 (50%)	1 (50%)
d8c98da8	2	2 (0%)	0 (100%)	0 (100%)	2 (0%)	2 (0%)

flattening can significantly increase the number of high-order nesting code blocks, which makes Wasm binaries unreadable and hard to follow.

We also measure the number of `call_indirect` after performing alias disruption obfuscation, which is shown in Table 3. We can easily observe that the number of `call_indirect` increases by 10.9 times after the obfuscation, while the length of the corresponding elem section increases by 8.8 times. Interestingly, there is no `call_indirect` instructions in cases in $\mathbf{D}_5$, because all its belonging cases are manually constructed [40]. After converting all their `call` instructions, we have got cases with more than 600 `call_indirect` instructions with identical semantics. It can be concluded that *alias disruption obfuscating method can effectively convert* `call` *to* `call_indirect`. *And the length of elem section increases by at least an order of magnitude, which can significantly increase the workload for malicious users because they have to consider all of them as possible callees for each* `call_indirect` *instruction.*

5.4.2 Against Malware Detector. Due to the superior performance of Wasm over JavaScript [25], many malicious users secretly inject cryptominers written in Wasm into the victim webpages, or deploy a miner on its owned pages. Once a user unintentionally visits these pages, cryptominers will be executed by utilizing the computing resources (e.g., CPU and hard disks) of the visitors to gain profits for the deployers. However, current malware detectors, e.g., VirusTotal, can effectively and efficiently identify these cryptominers' intentional malicious intents, and report them as malware [5]. One of the responsibilities of obfuscation is to hide the original intents of the given programs. To this end, we applied all obfuscating methods except for the Collatz-based one to 18 cases in $\mathbf{D}_3$. Table 4 shows the results of how many security vendors in VirusTotal can identify these malware.

As we can see, without any obfuscating methods, every cryptominer can be identified by at least two security vendors, and the one with id `dd8aabc0` can even be detected by 32 detectors. Interestingly, we can observe that code obfuscations ($\mathbf{OB}_3$ and $\mathbf{OB}_4$) has little contribution in hiding original intents, especially the control flow flattening. None

https://t.me/learningnets

Table 5. Consumed time on analyzing Wasm binaries of $\mathbf{D}_4$ and $\mathbf{D}_5$ with different obfuscating options for manticore and wasp, where O_1 and O_2 indicate the integration of Collatz-based opaque predicates, and TO refers to timeout (48 hours).

	manticore		**wasp**
	$\mathbf{D}_4$	$\mathbf{D}_5$	$\mathbf{D}_5$
original	0.28h	12.3h	0.5h
flattening	0.28h	12.3h	0.5h
flattening (O_1)	TO (0/159)	TO (0/32)	TO (0/32)
flattening (O_2)	TO (0/159)	TO (0/32)	TO (0/32)
alias disruption	0.28h	12.3h	0.5h
alias disruption (O_1)	TO (0/159)	TO (0/32)	0.6h
alias disruption (O_2)	TO (0/159)	TO (0/32)	0.75h

of these 18 cases can escape VirusTotal. Contrarily, name obfuscating methods perform very well. By simply applying name obfuscation and memory obfuscation independently, 2 and 11 cases have escaped all detectors in VirusTotal, respectively, accounting for 11.1% and 61.1% of all cases. Moreover, if we combine these two methods, all 18 cases cannot be identified by any detectors at all. To this end, we can conclude that most of detectors in VirusTotal identify malicious Wasm binaries by matching function names and string literals, instead of recovering semantics by static analysis.

5.4.3 Against Static Analysis. As we mentioned in §5.2, we choose manticore and wasp as representatives for static analyzers. To this end, the effectiveness of WASMixer against static analysis is equivalent to whether an obfuscated Wasm binary can be analyzed by symbolic executors within an acceptable range in terms of both consumed time and resources. In §4.2.3, we have introduced the Collatz-based opaque predicates that can be integrated into the control flow flattening and alias disruption. Both of these can deliberately introduce path explosion due to the characteristics of Collatz conjecture. Table 5 illustrates the consumed time on obfuscated Wasm binaries of cases in $\mathbf{D}_4$ and $\mathbf{D}_5$.

As we can see, the first column is the obfuscating methods adopted, where O_2 indicates a more aggressive obfuscation than O_1. For example, for the flattening, O_1 only inserts two Collatz-based opaque predicates for each function, while O_2 divides every ten instructions into a code block and appends a predicate. The following columns refer to consumed time, where timeout (TO) indicates the dataset cannot be finished within 48 hours. Comparing the 3rd, 4th and 7th row, we can see that adopting simple control flow flattening and alias disruption without integrating Collatz-based opaque predicates can only improve the unreadability (see §5.4.1) rather than the resistance against symbolic execution. However, even if adopting a less aggressive option (O_1), both manticore and wasp cannot solve even one of the cases from either $\mathbf{D}_4$ or $\mathbf{D}_5$ within two days at all. This is because once a function is called, the two Collatz predicates should be solved. However, the number of paths grows exponentially due to its inherent characteristic. Interestingly, for wasp, both the O_1 and O_2 Collatz-based alias disruption can only extend the consumed time by 20% and 50%, respectively. We think the reason behind such an ineffectiveness is twofold. First, the wasp is a concolic symbol executor, which is inherently more efficient than static symbol executors like manticore. Second, the cases of $\mathbf{D}_5$ are not directly compiled from source code written in high-level programing language, but generated by artificial construction, which significantly decreases the number of `call` instructions (as shown in Table 3). To this end, it has fewer instrumented points for inserting the Collatz opaque predicates. We have to argue that the manually constructed Wasm binaries are extremely unusual in real-world environments. Based on the results, we can conclude that *both the Collatz-based control flow flattening and alias disruption can effectively hinder the performance of symbolic executors.*

https://t.me/learningnets

Table 6. Overheads of consumed time and binary size brought by different obfuscating options on datasets, where T and BS refer to executing time and binary size, respectively, and O_1 and O_2 stand for identical meaning in §5.4.3.

	$\mathbf{D}_1$		$\mathbf{D}_2$		$\mathbf{D}_4$		$\mathbf{D}_5$	
	T	**BS**	**T**	**BS**	**T**	**BS**	**T**	**BS**
original	1	1	1	1	1	1	1	1
name obfuscation	0.96	1.00	0.98	0.96	1.01	1.00	0.99	1.02
memory obfuscation	1.11	1.22	1.47	1.04	1.14	1.24	0.99	1.00
flattening	1.02	1.09	1.08	1.01	1.08	1.14	1.01	1.17
flattening (O_1)	1.06	1.13	1.19	1.02	1.18	1.14	1.13	1.18
flattening (O_2)	1.11	1.14	1.21	1.03	1.27	1.24	1.29	1.36
alias disruption	1.03	1.06	1.09	1.00	1.07	1.02	1.00	1.02
alias disruption (O_1)	1.09	1.25	1.23	1.08	1.08	1.23	1.15	1.42
alias disruption (O_2)	1.28	1.26	1.28	1.08	1.28	1.23	1.19	1.43

RQ-2 Answer

After adopting different obfuscation methods on cases from $\mathbf{D}_1$ to $\mathbf{D}_5$, results show that WASMIXER can provide sufficient capabilities to resist manual reverse engineering, hide original intents, and hinder state-of-the-art static analyzers.

5.5 RQ3: Overhead

Except for standalone Wasm binaries, they are often uploaded and used in web browsers as a library, e.g., graphic computation [61] and cryptocurrency mining [14]. Therefore, users will be sensitive to their size and runtime performance, which are related to loading consumed time and executing time, respectively. As side effects of obfuscating, introducing overheads in terms of binary size and runtime are inevitable. But we should minimize their effects as small as possible. To evaluate them, we adopt all mentioned obfuscating options on all datasets except for $\mathbf{D}_3$, because of its strong dependency on environment (see §5.2). Table 6 illustrates the measured results.

As we can see, the name obfuscation nearly brings no overhead in both runtime and binary size. This is because name obfuscation totally targets on the function names in the custom section. For a virtual machine, whether the function name is readable has no impact on executing speed. We believe that fluctuations in the numbers at the 4th row are the result of measurement errors. As for the memory obfuscation, its imported overhead fluctuates significantly. Interestingly, we can observe that it has no impact on cases in $\mathbf{D}_5$. This is because they are constructed manually, for simplicity, authors do not introduce any interactions with the memory. For the other three datasets, we can easily conclude that the overhead brought by the memory obfuscation has different impact on consumed time and binary size. For example, cases in $\mathbf{D}_2$ interacted with the environment by command-line arguments, indicating lots of memory manipulation operations. Therefore, around 47% overhead is imported in terms of executing time. However, compared to its total number of instructions (tens of thousands instructions for each case), the overhead of binary size by hooking memory related instructions can be neglected.

https://t.me/learningnets

Moreover, as for code obfuscation, even under the most aggressive option, it will only import less than 30% overheads, except for the artificially constructed $\mathbf{D}_5$. From Table 5 in §5.4.3, we can conclude that the obfuscation effect of O_1 is enough for hindering the current state-of-the-art symbolic executors. Under this scenario, WASMixer can only introduce less than 20% overhead in both consumed time and binary size, which we think is a balance point between security and performance. Therefore, it is shown that *WASMixer can introduce less than 30% overhead in terms of binary size and executing time under the most aggressive obfuscating options. Considering the balance between efficiency and effectiveness, WASMixer can achieve a better performance.*

RQ-3 Answer

Name obfuscation brings no overhead, while memory obfuscation brings a higher overhead on command-line interacting Wasm binaries in terms of executing time. Moreover, considering the balance between effectiveness and efficiency, code obfuscation under O_1 brings around no more than 20% overhead in both terms of consumed time and binary size.

6 RELATED WORK

WebAssembly Binary. There are many works based on the Wasm binaries, covering various aspects, e.g., program analysis [17, 28, 35, 51, 54], malware detection and evasion [18, 47, 58], and testing [27, 37]. Specifically, Wasabi [54] is a dynamic analysis framework for Wasm which is able to perform instructions instrumentation on Wasm binaries. To obtain analysis results, Wasabi also inserts some helper functions as imported ones, which take the instrumented instructions as input. Moreover, to detect malicious cryptomining programs, SEISMIC [58] monitors the frequency of specific instructions in the given Wasm binary at runtime by instrumenting them. In addition, J.C. Arteaga et al. [18] implements Wasm-mutate to perform malware evasion on Wasm binaries. It provides many predefined strategies, e.g., instructions replacement and module structure transformation.

Code Obfuscation. Obfuscation has been widely adopted for decades to resist human reverse engineering and program analysis techniques. Lots of representative work were proposed before [9, 12, 19, 31, 33, 38, 46, 48, 50, 52, 59, 68]. For example, Proteus [9] adopts the virtual machine technology to perform obfuscation by translating the original program to a new instruction set used by the virtual machine. Tigress [52] is a source-to-source obfuscator for C. It supports many traditional obfuscation methods, e.g., control flow obfuscation, and virtualization. Due to the development of program analysis techniques, countering symbolic execution should be taken into consideration when implementing an obfuscator [12, 46, 48, 59]. Specifically, S. Banescu et al. [12] apply range dividers and input variants to deliberately increase the number of feasible paths, and M. Ollivier et al. [46] thoroughly study path-oriented protections to hinder symbolically executing. In addition, there are many efforts aimed at SMT solvers used in symbolic execution [48, 50, 71]. For example, M. Schloegel et al. [48] proposed a general framework for synthesizing MBA expressions that can effectively address current anti-obfuscation attacks. As for obfuscation on Wasm, there is only one work [13] which evaluated if source-level obfuscation can still evade malware detection on compiled Wasm binaries. However, they do not propose any obfuscator and our results mentioned in §5.4.2 are better than theirs.

7 THREATS OF VALIDITY

Deobfuscation. Pattern attack is a method that recognizes specific code structures generated by obfuscations and deobfuscates them. To avoid being deobfuscated, a common method is to increase the code diversity, which can be

https://t.me/learningnets

easily achieved on binary level. For example, in control flow flattening, we can insert dead code blocks in each layer to diversify the code structure. Moreover, to avoid the opaque predicates being identified, we can construct multiple variants based on our proposed Collatz-based one (like introducing hash function), and insert one of them before `call_indirect`.

Against Wasm, Binaryen [15] is the only usable tool to conduct deobfuscation to some extent. Specifically, it will lift the Wasm bytecode to its custom IR, perform optimizations, and compile it back to Wasm. Not surprisingly, according to our experimental results, Binaryen can deobfuscate around 70% obfuscation effect of control flow flattening and alias disruption by counting the number of flattened code blocks and `call_indirect` instructions. However, such a deobfuscation turns totally invalid when introducing Collatz-based opaque predicates to the above obfuscation methods. Such a situation is evaluated in §5.5 under the O_1 option. As we can see, the overhead in both terms of executing time and binary size increased only 14% and 18%, respectively, which is acceptable for a more robust obfuscation.

Potential obfuscation method. In addition to five proposed obfuscating methods, we emphasize that other obfuscating methods can be easily achieved through extending WASMixer. For example, the code block splitting and code block rearranging method mentioned in §4.2.1 can serve as the prerequisite for other control flow obfuscations, such as random control flow and bogus control flow. Moreover, rearranging these split code blocks can also achieve other obfuscating goals. Additionally, the *Mixed Boolean-Arithmetic* (MBA) [71] approach that is often used to resist symbolic execution can also be easily introduced in Wasm due to the stack-based format of Wasm. Once an MBA expression is converted to the AST format, it can be easily inserted into Wasm binaries.

Benchmark. When evaluating the effectiveness of WASMixer, we select different datasets for different jobs, which is mainly due to the limited functionalities of these tools. For example, although manticore can symbolically execute Wasm binaries, it does not support WASI (used by $\mathbf{D}_1$ and $\mathbf{D}_2$) and customized import functions ($\mathbf{D}_3$). Similarly, wasp can only support a customized set of instructions, i.e., $\mathbf{D}_5$. Moreover, we claim that these datasets are representative. Specifically, they cover both standalone ($\mathbf{D}_1$ and $\mathbf{D}_2$) and web-integrated format ($\mathbf{D}_3$). They are either compiled from source code ($\mathbf{D}_1$ to $\mathbf{D}_4$) or even composed manually ($\mathbf{D}_5$). Moreover, there are simple algorithms ranging from sequence searching to string operations ($\mathbf{D}_1$), and complicated real-world applications ($\mathbf{D}_2$). Therefore, we believe that these five datasets are representative enough, and are not designed specifically for evaluating tasks.

8 CONCLUSION

This paper presents the first general-purpose Wasm binary obfuscation framework, which works on both data-level and code-level obfuscation. Results show that WASMixer can not only keep the original semantics intact, but also be effective in resisting human reverse engineering, hiding original intents, and hindering the analysis from state-of-the-art static analyzers. Moreover, it brings in acceptable overhead in terms of both executing time and binary size. Our research has shed light on the promising direction of Wasm binary research, including Wasm code protection, Wasm binary diversification, and the attack-defense arm race of Wasm binaries.

REFERENCES

[1] 2023. AutoCAD Web App. https://web.autocad.com/
[2] 2023. Collatz Conjecture. https://en.wikipedia.org/wiki/Collatz_conjecture
[3] 2023. List of WebAssembly Games. https://www.webassemblygames.com/
[4] 2023. Photoshop's journey to the web. https://web.dev/ps-on-the-web/
[5] 2023. VirusTotal official website. https://en.wikipedia.org/wiki/LEB128
[6] Bytecode Alliance. 2023. Github wasm-tools repository. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate

https://t.me/learningnets

[7] Ambient. 2023. Github Ambient repository. https://github.com/AmbientRun/Ambient

[8] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001.

[9] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. 2006. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*. 47–58.

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

[11] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. 2016. Control flow obfuscation for android applications. *Computers & Security* 61 (2016), 72–93.

[12] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200.

[13] Shrenik Bhansali, Ahmet Aris, Abbas Acar, Harun Oz, and A Selcuk Uluagac. 2022. A first look at code obfuscation for webassembly. In *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 140–145.

[14] Weikang Bian, Wei Meng, and Yi Wang. 2019. Poster: Detecting webassembly-based cryptocurrency mining. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2685–2687.

[15] Binaryen. 2023. Github Binaryen repository. https://github.com/WebAssembly/binaryen

[16] Blazor. 2023. Blazor official webpage. https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor

[17] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* 118 (2022), 102745.

[18] Javier Cabrera-Arteaga, Martin Monperrus, Tim Toady, and Benoit Baudry. 2022. WebAssembly Diversification for Malware Evasion. *arXiv preprint arXiv:2212.08427* (2022).

[19] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A taxonomy of obfuscating transformations*. Technical Report. Department of Computer Science, The University of Auckland, New Zealand.

[20] Collections-C. 2023. Github Collections-C repository. https://github.com/srdja/Collections-C

[21] collections-c-for-gillian. 2023. Github collections-c-for-gillian repository. https://github.com/GillianPlatform/collections-c-for-gillian

[22] deepMiner. 2023. Github deepMiner repository. https://github.com/deepwn/deepMiner

[23] Emscripten. 2023. Emscripten official website. https://emscripten.org/

[24] eosio. 2023. eosio official website. https://eos.io/

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

[26] Adam Hall and Umakishore Ramachandran. 2019. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*. 225–236.

[27] Keno Haßler and Dominik Maier. 2021. Wafl: Binary-only webassembly fuzzing with fast snapshots. In *Reversing and Offensive-oriented Trends Symposium*. 23–30.

[28] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. EOSAFE: Security Analysis of EOSIO Smart Contracts.. In *USENIX Security Symposium*. 1271–1288.

[29] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the Web Conference 2021*. 2696–2708.

[30] Ting-Wei Hou, Hsiang-Yang Chen, and Ming-Hsiu Tsai. 2006. Three control flow obfuscation methods for Java software. *IEE Proceedings-Software* 153, 2 (2006), 80–86.

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

[32] KLEE. 2023. KLEE official website. https://klee.github.io/

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

[34] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything old is new again: Binary security of webassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 217–234.

[35] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Pecise Types from WebAssembly Binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 410–425.

[36] Daniel Lehmann, Michelle Thalakottur, Frank Tip, and Michael Pradel. 2023. That'sa Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly. In *Symposium on Software Testing and Analysis (ISSTA'23)*.

[37] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly. *arXiv preprint arXiv:2110.15433* (2021).

https://t.me/learningnets

[38] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*. 290–299.

[39] Niko Mäkitalo, Tommi Mikkonen, Cesare Pautasso, Victor Bankowski, Paulius Daubaris, Risto Mikkola, and Oleg Beletski. 2021. WebAssembly modules as lightweight containers for liquid IoT applications. In *Web Engineering: 21st International Conference, ICWE 2021, Biarritz, France, May 18–21, 2021, Proceedings*. Springer, 328–336.

[40] Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. 2022. Concolic Execution for WebAssembly. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[41] MDN. 2023. MDN web docs website. https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_wasm

[42] MDN. 2023. MDN web docs website. https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API

[43] MDN. 2023. Understanding the text format. https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

[44] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[45] obfuscation-benchmarks. 2023. Github obfuscation-benchmarks repository. https://github.com/tum-i4/obfuscation-benchmarks/tree/master/basic-algorithms

[46] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to Kill Symbolic Deobfuscation for Free. (2019).

[47] Alan Romano, Yunhui Zheng, and Weihang Wang. 2020. Minerray: Semantics-aware analysis for ever-evolving cryptojacking detection. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1129–1140.

[48] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*. 3055–3073.

[49] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[50] Toshiki Seto, Akito Monden, Zeynep Yücel, and Yuichiro Kanzaki. 2019. On preventing symbolic execution attacks by low cost obfuscation. In *2019 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. IEEE, 495–500.

[51] Quentin Stiévenart, David W Binkley, and Coen De Roover. 2022. Static stack-preserving intra-procedural slicing of webassembly binaries. In *Proceedings of the 44th International Conference on Software Engineering*. 2031–2042.

[52] tigress. 2023. tigress official website. https://tigress.wtf/index.html

[53] TinyGo. 2023. TinyGo official docs webpage. https://tinygo.org/docs/guides/webassembly/

[54] Andres Veidenberg, Alan Medlar, and Ari Löytynoja. 2016. Wasabi: an integrated platform for evolutionary sequence analysis and data visualization. *Molecular biology and evolution* 33, 4 (2016), 1126–1130.

[55] wabt. 2023. wasm2wat tool website. https://webassembly.github.io/wabt/doc/wasm2wat.1.html

[56] wabt. 2023. wat2wasm tool website. https://webassembly.github.io/wabt/doc/wat2wasm.1.html

[57] Dong Wang, Bo Jiang, and WK Chan. 2020. WANA: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection. *arXiv preprint arXiv:2007.15510* (2020).

[58] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II 23*. Springer, 122–142.

[59] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. Springer, 210–226.

[60] wapm. 2023. wapm official website. https://wapm.io/

[61] wasm-webgpu. 2023. Github wasm-webgpu repository. https://github.com/juj/wasm_webgpu

[62] wasp. 2023. Github wasp repository. https://github.com/wasp-platform/wasp/tree/main/wasp/tests

[63] webasm. 2023. webasm official webpage. https://www.mainconcept.com/webasm

[64] WebAssembly. 2023. Github wasi-sdk repository. https://github.com/WebAssembly/wasi-sdk

[65] WebAssembly. 2023. Name section format in WebAssembly specification. https://webassembly.github.io/spec/core/appendix/custom.html#name-section

[66] WebAssembly. 2023. WebAssembly specification webpage. https://webassembly.github.io/spec/core/appendix/index-instructions.html

[67] WebAssembly. 2023. WebAssembly staic validation algorithm. https://webassembly.github.io/spec/core/appendix/algorithm.html#algo-valid

[68] Hui Xu, Yangfan Zhou, Yu Kang, Fengzhi Tu, and Michael Lyu. 2018. Manufacturing resilient bi-opaque predicates against symbolic execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 666–677.

[69] Alon Zakai. 2011. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 301–312.

[70] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. 25–36.

https://t.me/learningnets

[71] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. 2007. Information hiding in software with mixed boolean-arithmetic transforms. In *Information Security Applications: 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers 8*. Springer, 61–75.

https://t.me/learningnets