When Does Wasm Malware Detection Fail? A Systematic Analysis of Their Robustness to Evasion

Taeyoung Kim*, Sanghak Oh*, Kiho Lee†, Weihang Wang‡,
Yonghwi Kwon§, Sanghyun Hong¶, Hyoungshick Kim*
*Department of Electrical and Computer Engineering, Sungkyunkwan University, Republic of Korea,
{tykim0402, sanghak, hyoung}@skku.edu
†Electronics and Telecommunications Research Institute (ETRI), Republic of Korea, kiho@etri.re.kr
‡Department of Computer Science, University of Southern California, USA, weihangw@usc.edu
§Department of Electrical and Computer Engineering, University of Maryland, USA, yongkwon@umd.edu
¶Department of Computer Science, Oregon State University, USA, sanghyung.hong@oregonstate.edu

Abstract—**WebAssembly (Wasm) provides a language-agnostic compilation target that delivers near-native performance for web applications, yet it also attracts adversaries who exploit Wasm to effectively steal someone else's computer resources such as cryptojackers. While several detection tools have been proposed, their robustness against perturbations remains largely unknown.**

In this paper, we introduce SWAMPED (Systematic WebAssembly Module Perturbation Evaluation of Detectors), a framework that incorporates 22 semantics-preserving perturbation methods. SWAMPED generates a total of 48,840 perturbed variants from 43 cryptojacker samples and 31 additional Wasm malware binaries from real-world. We assess detection performance of six detectors: three Wasm-specific ones and three deep neural network (DNN) detectors. We find that DNN-based detectors are vulnerable to perturbations that shift the instruction distribution; profiling-based methods are disrupted by changes in instruction frequency; and semantic-aware approaches are highly sensitive to function-level dependency modifications. DNN-based detectors, which lack Wasm-specific modeling, are particularly susceptible to changes in the spatial layout of Wasm binaries. These findings highlight fundamental limitations in current Wasm malware detection approaches, relying on overly specific detection heuristics and inadequately trained or designed models. We offer suggestions to improve the robustness against perturbations.

I. INTRODUCTION

WebAssembly (Wasm) [1] is a language-agnostic compilation target that enables languages like C/C++ and Rust to execute efficiently on the web. It enhances the performance of web applications, making it possible to run computationally intensive tasks such as video editing and cryptographic operations. For instance, small programs in PolyBench C benchmarks [2] run over $8\times$ faster in Wasm than JavaScript [3].

Unfortunately, this capability has also attracted adversaries who create malicious web programs (*e.g.*, malicious advertisements, keyloggers, and cryptomining) [4]. Wasm's superior performance over JavaScript has made it particularly popular for cryptojacking, where adversaries exploit victims' computing resources to mine cryptocurrencies without consent. Wasm is also appealing for malicious use because it complicates program analysis and detection [5] as binaries are distributed in compiled form, unlike JavaScript [6], which is deployed as human-readable source code.

In response, researchers have proposed detection methods for Wasm-based malware [7]–[13], leveraging techniques from traditional program analysis to machine learning. While effective against known malware samples, their preparedness for real-world deployment with diversified malware variants remains unknown. Recent work demonstrates that adversaries can use automated diversification techniques [14], [15] to generate numerous malware variants at negligible cost, potentially undermining existing detection tools. While a few studies have examined perturbation impacts on malware detectors [16], [17], a systematic study of detection method limitations under automated diversification is still missing. Moreover, Wasm's unique structure and environment (*e.g.*, stack-based execution model, structured control flow, and inter-section dependencies) [18] affect the feasibility and effectiveness of the perturbations, making prior findings on binary obfuscation difficult to apply to Wasm [19]–[22] (see Section V).

In this paper, we systematically study how perturbation-based malware diversification in the Wasm context impacts various Wasm malware detection approaches. Specifically, we analyze the underlying causes of detection failures and offer insights into potential improvements. To this end, we develop a framework, SWAMPED (Systematic WebAssembly Module Perturbation Evaluation of Detectors), which incorporates 22 semantic-preserving perturbation strategies for Wasm binaries. SWAMPED assesses the impact of each perturbation on 43 real-world cryptojacker samples against three state-of-the-art detectors: Minos [7], MineSweeper [8], and MinerRay [9], as well as 31 real-world malware samples against three DNN-based detectors: WasmGuard [12], MalConv [23], and AvastNet [24].

Our evaluation reveals key weaknesses in these detectors. Minos [7] fails under *distribution-shifting instruction insertions*, revealing sensitivity to structural patterns. MineSweeper [8] is evaded by small changes to cryptographic opcode frequencies, showing reliance on static thresholds. MinerRay [9] breaks under control/data-flow perturbations that fragment semantic paths. Even among DNN-based techniques, specialized/tuned techniques such as WasmGuard [12] are prone to fail on instruction-level perturbations, while Mal-

Conv [23] and AvastNet [24] fail on both code and non-code perturbations. More importantly, while detectors may perform similarly in flagging malware, their reactions to perturbed samples vary, often reflecting their subtle underlying system or model constructions, demonstrating the importance of understanding their performance under perturbation.

Our contributions are summarized as follows:

- We develop 22 perturbation methods applicable to Wasm-based malware according to Wasm's binary structure.
- We analyze the impact of the perturbation methods on state-of-the-art Wasm malware detection tools, evaluating detection performance across 48,840 perturbed variants.
- We have released our framework as open-source: https://github.com/SKKU-SecLab/SWAMPED.

II. BACKGROUND

A. WebAssembly (Wasm)

WebAssembly is a W3C standard offering a high-performance, language-agnostic compilation target with enhanced security. Programs written in C, C++, and Rust compile into Wasm binaries that execute at near-native speeds within a browser's virtual machine. A Wasm module consists of functions, global variables, linear memory, and an optional table for indirect function calls, all identified by integer indices.

Stack-based Virtual Machine. WebAssembly operates on a stack-based virtual machine where instructions implicitly manipulate the values on the stack. Besides the stack, there are local and global variables; locals are scoped to their declaring function, while globals are accessible throughout the module.

Data Types. Variables and function signatures in Wasm are strictly typed, using 4 primitive data types: 32/64-bit integers and floating-point numbers (i32, i64, f32, and f64). Complex data structures are compiled down to these primitives [25].

Control Flow. Wasm uses a block-based control flow mechanism. A block contains instructions that do not change the control flow. Branching occurs only at block ends, and no inter-procedure branching or mid-block jumps are allowed. Multi-way branching uses branch tables (br_table) with predefined targets. Indirect function calls use call_indirect, retrieving a function index from the stack to look up the target. This structure prohibits arbitrary jumps and data execution as code, limiting control flow manipulation.

Memory Management. Memory in WebAssembly is linear and unmanaged—a contiguous array of bytes accessible via load and store instructions. Programs can request additional memory at runtime [18] and memory management functions are typically implemented within the Wasm module itself.

B. Wasm Malware

In this section, we discuss Wasm's binary format and execution environment, which significantly impact the development of web malware as well as malware detection techniques.

Efficient Execution Environment. Before Wasm, JavaScript was the primary language for implementing complex logic on the web. While widely supported, JavaScript suffers from performance limitations, particularly for computation-intensive tasks. This posed challenges for implementing operations like encryption, decryption, or hashing efficiently in the browser. One notable case is cryptomining. As cryptojacking emerged as a profitable malware tactic, attackers initially turned to JavaScript to implement cryptominers in websites. However, due to JavaScript's limited performance, they often failed to generate meaningful profit [8], [26].

The introduction of Wasm significantly changed this landscape. Wasm enables near-native execution speed for computation-heavy tasks while maintaining compatibility with modern browsers. This performance enhancement made it feasible to implement efficient, browser-based cryptominers.

Wasm Binary Format. Wasm programs are deployed as compiled Wasm binaries, unlike JavaScript programs, which are deployed as their source code. This imposes multiple challenges in analysis. First, the binary format itself is designed for efficient execution, not readability, making it harder to interpret manually compared to JavaScript source code. It requires analysis techniques that can handle binaries. Second, during the compilation, high-level semantics in the source code (*e.g.*, data structures, data types, and control structures) are significantly changed or stripped away. Third, Wasm binaries can be generated from various languages, such as C and Rust, where compiler optimizations and instrumentation methods are available, making the analysis of the compiled binary more challenging. Fourth, the binary format makes automated code obfuscation/perturbation [15], [27], [28] applicable, imposing an additional challenge.

Challenges in Wasm Analysis and Malware. Wasm's binary format and compilation process strip high-level semantics, hindering manual program analysis and making it attractive for malware deployment compared to JavaScript. The limited availability of mature Wasm-specific analysis tools leaves security systems vulnerable to malicious behavior inside Wasm modules. Unlike JavaScript, which has established linters, de-obfuscators, and static analyzers, Wasm lacks comprehensive tooling support. This gap enables malicious actors to use Wasm for delivering payloads and executing malicious logic, with many malicious binaries going undetected. Furthermore, since Wasm binaries are generated from languages like C/C++ that support various obfuscation and encryption techniques, attackers can easily apply these methods to complicate analysis.

C. Wasm Malware Detectors

There is a line of research focusing on detecting malicious Wasm binaries and programs [7]–[9], [11], [12], [29]–[35]. These approaches span a range of techniques, including instruction frequency analysis [8], [30], [33], behavior modeling [9], [11], [34], dynamic profiling [29], [31], [32], [35], and deep learning-based classification [7], [12].

Among the twelve relevant techniques [7]–[9], [11], [12], [29]–[35], six techniques [11], [31]–[35] do not provide open-source systems, hence are excluded. Outguard [29] is excluded as it does not analyze the Wasm binary's content. Instead, it only checks the presence of Wasm modules, making any perturbation on them irrelevant. MineThrottle [30] requires
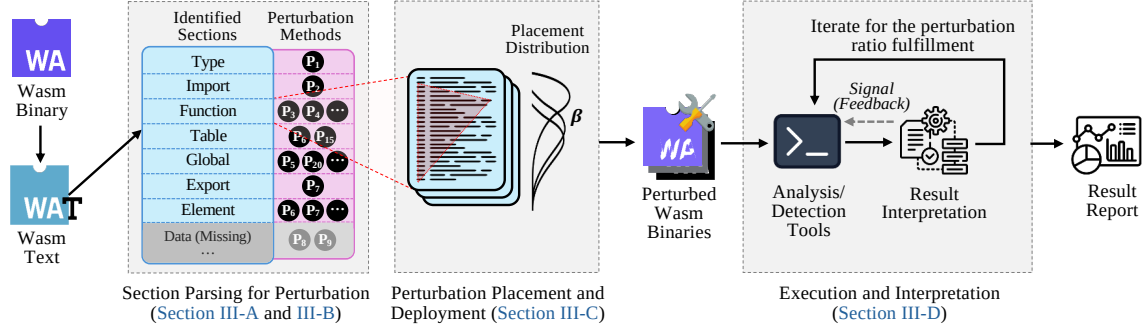
Fig. 1. SWAMPED workflow: It perturbs WebAssembly binaries and evaluates the robustness of malware detection tools.

JavaScript modules for its analysis and is unable to analyze Wasm modules independently; hence, it is also excluded.

We identify four open-source Wasm malware detectors that directly analyze Wasm binaries: Minos [7], MineSweeper [8], MinerRay [9], and WasmGuard [12]. Additionally, to broaden our evaluation scope, we adapt two widely-studied DNN-based malware detectors originally designed for Windows PE files—MalConv [23] and AvastNet [24]—to work with Wasm binaries. These general-purpose detectors have been extensively used in static malware classification and adversarial malware generation [14], [36], [37], and their inclusion allows us to assess the broader applicability of our perturbation techniques beyond Wasm-specific detectors. We train binary classifiers using their architectures to distinguish between benign and malicious Wasm binaries. Table I summarizes the characteristics of all six detectors.

Detector	Base Technique	Detection Method	Granularity
Minos [7]	2D CNN classifier	Image recognition	Program
MineSweeper [8]	Operation profiling	Instruction frequency	Function
MinerRay [9]	Semantic-aware analysis	IR & CFG analysis	Function
WasmGuard [12]	Adversarial training	Adversarial classification	Program
MalConv [23]	Gated CNN classifier	Byte-level pattern learning	Program
AvastNet [24]	Deep CNN classifier	Byte-level hierarchical features	Program

- **Minos [7]** converts Wasm binaries into grayscale images and trains a convolutional neural network (CNN) classifier to identify static patterns. A Wasm binary is first converted into an array of integers, where each integer represents a pixel in the grayscale image.
- **MineSweeper [8]** analyzes CryptoNight-based mining algorithms by profiling Wasm operations in cryptographic functions (*e.g.*, BLAKE, Keccak, and AES). It generates fingerprints based on the operation frequency per function to identify characteristics of cryptomining.
- **MinerRay [9]** is a semantic-aware static analysis-based technique. It analyzes the semantics of cryptomining algorithms across both Wasm and JavaScript contexts. Specifically, it converts JavaScript into Wasm and then combines it with the original Wasm binary. The binary

is abstracted into an intermediate representation (IR) to construct a control flow graph (CFG), which is used to identify the semantics.

- **WasmGuard [12]** is a detector against malicious Wasm binaries designed to be resilient to adversarial attacks. It leverages adversarial training based on the Fast Gradient Sign Method (FGSM) [38] and incorporates adversarial contrastive learning. It creates adversarial samples by inserting values from the FGSM model. The values are injected into newly created custom sections. In the existing sections (*e.g.*, type and function), dummy elements are also injected.
- **MalConv [23]** is a shallow CNN that processes byte sequences through dual parallel convolution paths for feature extraction and gating. The outputs are fused via element-wise multiplication and followed by temporal max-pooling and a fully connected classifier.
- **AvastNet [24]** is a deeper CNN classifier with four stacked convolutional layers with increasing strides and interleaved max-pooling layers to downsample long byte sequences, followed by global average pooling and multiple fully connected layers.

III. OUR ANALYSIS FRAMEWORK: SWAMPED

We introduce SWAMPED (Systematic WebAssembly Module Perturbation Evaluation of Detectors), a framework for assessing the robustness of Wasm-based malware detection tools. SWAMPED automatically applies fine-grained perturbations to Wasm binaries and evaluates their impact on detection performance. The framework generates a comprehensive report, highlighting the most effective perturbations against each tool and identifying related evasion techniques.

Workflow Overview. Figure 1 illustrates the overall workflow of SWAMPED. First, it converts a Wasm binary into the Wasm text format using the wasm2wat [39] tool with the --generate-names flag, which generates symbolic names for section entries to serve as identifiers during parsing. This conversion enables parsing of different Wasm binary sections, such as Type, Import, Function, Table, Global, Export, Element, and Data sections. Note that these generated names are temporary and are not preserved in the final perturbed binary.

Then, for each section, SWAMPED identifies the applicable perturbation methods for the target. For each applicable pertur-

bation, if there are multiple targets (*e.g.*, multiple instructions that can be perturbed), we further select which elements to be perturbed. To make a statistically meaningful placement of perturbations, SWAMPED uses the beta distribution to select perturbation positions (*i.e.*, what to perturb). SWAMPED then applies the perturbation to create a perturbed Wasm binary.

All the perturbed binaries are then fed into various Wasm analysis and detection tools, including Minos, MineSweeper, MinerRay, WasmGuard, MalConv, and AvastNet. These tools classify the perturbed binary as Benign, Malign, Bad, Suspect, Exact, Unlikely, Probable, or Certain, depending on their detection algorithms, often accompanied by a confidence score. The entire process is iteratively repeated for a range of perturbation ratios (from 10% to 100%), until the desired perturbation intensity is fulfilled. SWAMPED summarizes the detection outcomes across all perturbation types and ratios, generating metrics such as evaded sample rate and average perturbation ratio among evasive samples.

A. Section Parsing

SWAMPED first parses the target binary to identify all sections. As shown in Figure 2, Wasm binaries have a hierarchical structure with eight main sections: Type, Import, Function, Global, Export, Element, Data, and Custom. For each identified section, SWAMPED determines the applicable perturbation methods (third column of Figure 2), which are detailed in Section III-B.

Section	WebAssembly Code	Our Perturbation
Type Section	type 0: func (param i32) (result i32) type 1: func (param i64) ...	P1
Import Section	"import_func" (func_0 (type 0))	P2

Function Section	func_1 (type 1) (param i64):
	    local.get 0
	    ...
		P3 P4 P10 P12 P13 P14 P15 P16 P17 P18 P19 P20 P21 P22

Table Section	global 0: (mut i32) (i32.const 0) ...	P6 P13
Global Section	global 0: (mut i32) (i32.const 0) ...	P5 P20 P22
Export Section	export "export_func" (func_1) ...	P7
Element Section	element 0: (i32.const 0) -> 100 ...	P6 P9 P15
Data Section	data 0: (i32.const 1024) "\010203" ...	P4 P8
Custom Section	"\010203" (optional info)	P10

Fig. 2. Wasm binary's section structure.

The Type section defines function signatures, such as 'func (param i32) (result i32).' The Import section declares imported functions. The Function section contains function bodies (*i.e.*, functions' code blocks). The Table section defines function reference tables. The Global section declares global variables. The Export section specifies functions to be exported by referring to those in the Function section. The Element and Data sections contain initial data/constant values for variables (*e.g.*, table entries) and memory. The Custom section holds optional metadata such as debugging information. Each of these sections comprises a sequence of entries.

B. Perturbation Methods

Given the target binary's sections identified in Section III-A, all associated (and thus applicable) perturbations to the identified sections (shown in the third column of Figure 2) are selected and applied later. All 22 perturbations of SWAMPED are semantics-preserving and grouped into two categories: *Structural Perturbation* and *Code Perturbation*. *Structural Perturbation* modifies non-code (or non-instruction) elements of the binary, such as data types or function definitions. *Code Perturbation*, by contrast, transforms the program code/logic of the target program into another form with the same semantics. Both types of perturbations preserve the program's original semantics and do not alter its behavior.

These perturbations are systematically derived from three complementary sources: *i)* 11 Wasm-specific techniques adapted from prior literature [12], [15], [28] with refined implementations (P1~P3, P5~P8, P10, P13, P15, P17), *ii)* 8 cross-platform techniques from C/LLVM [40]–[42] and PE [14], [43]–[45] contexts reinterpreted for Wasm's strict stack-based model (P4, P9, P11, P12, P14, P16, P20, P22), and *iii)* 3 new Wasm-specific transformations exploiting syntactic flexibility (P18, P19, P21). We aim for broad coverage, ensuring at least one perturbation per major Wasm section and instruction category. We excluded binary obfuscations that are not compatible with Wasm's execution and validation model, such as virtualization, packing, signature removal, and overlay injection.

1) Structural Perturbation (Non-code): SWAMPED has 10 perturbations methods that perturb structural data/elements of the target binary file such as function definitions or initial values of variables.

Type Section. This section contains function signatures that are essentially the definitions of functions.

P1 *Function Signature Insertion*: SWAMPED inserts a new function signature in the type section. Specifically, SWAMPED creates the new function signature by picking an existing function signature (based on our selection criteria, *i.e.*, a beta distribution, described in Section III-C) and randomly adding a local or parameter variable. The type of added variable is also randomly chosen. Note that if the resulting signature is identical to one of the existing function signatures, we repeat the process until a unique function signature is generated.

Import Section. This section contains elements representing items (*e.g.*, functions and variables) that it imports from others.

P2 *Import Insertion*: SWAMPED inserts a new import entry by sampling function names from a curated pool of 4,125 candidates. This pool comprises 80% entries extracted from benign Wasm binaries [8], [46], [47] and 20% synthetically generated entries to ensure diversity. To maintain runtime compatibility, we ensure that the corresponding function name is defined in the external environment (*e.g.*, JavaScript) to prevent linking errors due to unresolved imports.

Function Section. This section contains each function's code.

P3 *Function Insertion*: SWAMPED inserts a new function with its corresponding code implementation. The function code is randomly selected from a dataset of benign Wasm binaries [8], [46], [47], ensuring semantic preservation by

avoiding perturbation of the inserted function's logic.[1] To maintain binary consistency, we select only functions whose signatures match existing entries in the target binary's Type section, thereby reusing established function signatures and avoiding the need for additional type definitions.

P4 *Function Body Cloning*: SWAMPED clones an existing function f to create a functionally equivalent duplicate f_{clone} with a distinct function identifier. SWAMPED then redirects a subset of f's call sites to invoke f_{clone}, distributing the call graph while preserving program semantics. This perturbation targets detectors that rely on function body patterns or call graph analysis for malware identification.

Global Section. An entry in the global section represents the name and value of a global variable.

P5 *Global Insertion*: SWAMPED inserts a new global entry by randomly selecting a data type from the four primitive Wasm types (i32, i64, f32, f64) and generating an associated random initial value within the type's valid range.

Element Section. An entry in the element section specifies function references used to initialize a table, which are subsequently used for indirect function calls.

P6 *Element Insertion*: SWAMPED creates a new element entry containing a valid function reference index (*e.g.*, i32.const 9) paired with a corresponding function identifier (*e.g.*, F56)[2]. To maintain isolation from the existing table structure in the target binary, we define a separate table specifically for these newly inserted entries, preventing interference with the original table's functionality.

Export Section. This section defines the items (*e.g.*, functions and variables) that the module makes available for others.

P7 *Export Insertion*: SWAMPED inserts a new export entry by sampling function names from a curated pool of 26,000 candidates, consisting of 80% entries extracted from benign Wasm binaries [8], [46], [47] and 20% synthetically generated entries. To preserve semantic consistency, we ensure the selected export function references an already-defined function in the target binary, maintaining consistent function visibility without introducing undefined references.

Data Section. An entry in the Data section provides initial values for regions of Wasm linear memory, with each entry initializing memory sequentially in the order it appears.

P8 *Data Insertion*: SWAMPED employs a systematic approach to data insertion by first scanning existing data entries to construct a shadow memory map that reflects the final memory layout after initialization. SWAMPED then samples a memory slice with a randomized offset and length from this map, using it as the foundation for a new data entry. This method ensures the initialized memory state remains intact and that no existing data is corrupted during insertion.

P9 *Data Encryption*: SWAMPED implements data obfuscation by replacing the data value of an existing entry with its XOR-encrypted form. To maintain runtime correctness, a corresponding decryption function is automatically inserted into the binary and registered in the Start section, ensuring immediate execution upon Wasm module instantiation and transparent restoration of original values.

Custom Section. This section provides a flexible mechanism to embed additional, non-standard data within a Wasm binary.

P10 *Custom Section Insertion*: SWAMPED inserts new custom sections by leveraging the wasm-objdump [39] tool to identify section boundaries within the raw binary, since custom sections are not preserved during Wasm text format conversion. The insertion process selects a random location between existing sections and populates the new section with byte content sampled from raw bytes of benign Wasm binaries [8], [46], [47], maintaining structural integrity while introducing benign variations.

2) Code Perturbation: SWAMPED provides 12 semantic-preserving code transformations on function code.

No Operation Instruction. WebAssembly supports No-operation (NOP) instruction, which performs no computation and hence does not affect any program semantics.

P11 *NOP Insertion*: SWAMPED inserts a NOP instruction at a chosen location (see Section III-C). Since NOP has no dependencies, it has no constraints for the insertion location.

Stack Operation Instruction. Stack operation instructions consume values from the stack and push the result onto the stack. They are fundamental in Wasm as it runs a stack-based virtual machine. As those instructions change the stack, when perturbing stack operation instructions, we often add additional instructions to prevent undesirable changes in the stack.

P12 *Stack Operation Insertion*: To insert a stack operation, SWAMPED adds a pair of instructions that push values to the stack (*e.g.*, defining constant values) and instructions that pop values from the stack (*e.g.*, shl pops its arguments from the stack). As some instructions push the results back to the stack, SWAMPED adds additional instructions to discard them (*e.g.*, drop discards a value from the stack). For example, the shift-left (shl) instruction is paired with two constant value definitions for the instruction's arguments as well as a drop instruction to clean up the result pushed by the shl instruction.

SWAMPED excludes instructions such as store that operate memory space outside of the stack, as they require SWAMPED to change the memory layout of the target program (*i.e.*, adding a new memory buffer). Instructions making irreversible changes to the memory state, such as grow (increasing the allocated buffer's size)[3], are also excluded.

Opaque Predicate. Opaque predicates [48], [49] are predicates whose conditions are known to developers but difficult for analysis techniques to infer at runtime. They are widely used in software obfuscation [50], [51] to hinder static analysis by obscuring predicate conditions.

P13 *Opaque Predicate Insertion*: SWAMPED uses the Collatz conjecture [52] to generate opaque predicates. When choosing locations to insert the opaque predicates, Note that SWAMPED

---

[1]We pruned out 60,636 functions (out of 66,378) that have dependencies (*e.g.*, ones calling another user-defined function) and replaced all global variable accesses with local variables of the same data type.

[2]An inserted entry would be 'elem (table T2) (i32.const 9) F56.'

[3]The change is irreversible; no instruction can shrink the expanded memory.

makes it configurable to disable inserting opaque predicates in loops[4] due to its impact on performance.

Function Call Instruction. Wasm supports two types of function calls: Direct (`call`) and indirect (`indirect_call`). `call` directly takes a callee function's identifier as an operand, while `indirect_call` takes a callee function's signature (*i.e.*, the type identifier of the callee) and a target function's reference index that is resolved through the `Element` section's function table.

(P14) *Proxy Function Insertion*: A proxy function is a wrapper function that calls its original function. For example, given a function f, its proxy function is a function f_{proxy} calling f and returning the return values from the call.

To implement this perturbation, SWAMPED picks an existing function f and creates a proxy function f_{proxy}. SWAMPED then changes the call targets of all the callers of f to f_{proxy}. This perturbation may affect analysis techniques/systems with improper (or incomplete) inter-procedure analysis.

(P15) *Direct to Indirect Call Transformation*: SWAMPED converts a direct call into an indirect call. Given a `call`, SWAMPED extracts the call target's type and inserts an identifier of the function in the `Element` section. SWAMPED then adds instructions setting up the arguments for the indirect call and `call_indirect` with the extracted type.

Listing 1 shows an example. `call` at line 3 is replaced with `call_indirect` at line 5. It takes the function's identifier `F75` by pushing a constant number 0 (at line 4), which means the first function element in the `Element` section at line 9 (`F75` in `NEW_TABLE` for `funcref`). It also takes the type of the function, which is `type T5` at line 5. Observe that we define a new `funcref` table at line 7 to avoid modifying the original table. The function index list size (*i.e.*, 1 at line 7) is adjusted to match the number of functions listed in the `Element` section. This perturbation imposes challenges to the systems that do not properly handle indirect calls.

```
1       i32.const 100
2       i32.const 200
3 [-]   call F75
4 [+]   i32.const 0
5 [+]   call_indirect (type T5)
6       ...
7 [+](table NEW_TABLE 1 funcref)
8       ...
9 [+](elem NEW_ELEM (table NEW_TABLE) func F75)
```

Listing 1. Converting a call operation into an indirect call operation.

Instruction Substitution. SWAMPED includes four types of instructions that can be replaced with a set of semantically equivalent instructions.

(P16) *Add/Sub Operation Transformation*: An addition (`add`) operation can be replaced with a subtraction (`sub`) by negating its operand. The reverse transformation also holds. For instance, '`i32.add (local.get x) (i32.const 20)`' becomes '`i32.sub (local.get x) (i32.const -20)`'

(P17) *Shift Operation Transformation*: Shift-left and -right (`shl` and `shr`) operations can be converted to multiplication (`mul`) and division (`div`) operations, respectively. Specifically, the operand of a shift operation is converted to the exponent of 2 in `mul` or `div` (*e.g.*, '`shl` with 3' → '`mul` with 2^3').

[4]In this work, it is disabled, meaning that it avoids loops.

(P18) *Eqz Operation Transformation*: `eqz` can be converted into an explicit equality check: '`i32.eq (i32.const 0)`.'

(P19) *Offset Expansion*: Offsets of memory operations (*e.g.*, `load offset=X`) are expanded into a set of instructions computing the offset. For example, '`i32.load offset=10`' can become '`i32.const 10; i32.add; i32.load`.'

Instruction Expansion via Mixed-boolean Arithmetic. Instructions such as `xor` and `or` can be expanded into longer expressions using Mixed-boolean Arithmetic (MBA) expressions, which are often used in software obfuscation [53], [54].

(P20) *Transforming XOR/OR to MBA*: For example, an `xor` operation can be transformed using the MBA expression $x \oplus y = 2 \cdot (x|y) \cdot y - x$, as described in previous work [55]. Instead of `xor` ($\oplus$), the MBA expression uses multiplication ($*$), or ($|$), and subtraction ($-$).

```
1       ...
2       i64.const 10
3       i64.const 20
4 [-]   i64.xor
5 [+]   global.set X
6 [+]   global.set Y
7 [+]   i64.const 2
8 [+]   global.get X
9 [+]   global.get Y
10 [+]  i64.or
11 [+]  i64.mul
12      ...
13 [+]  i64.sub
14      ...
15 [+]  (global X (mut i64) (i64.const 0))
16 [+]  (global Y (mut i64) (i64.const 0))
```

Listing 2. Converting an `xor` into mixed-boolean arithmetic operations.

Listing 2 shows how this transformation is implemented. First, the original operands x and y of the `xor` operation are stored to global variables (`$x` and `$y` in lines 15–16) and fetched accordingly (lines 5–6). Second, the substituted MBA expression's operations are sequentially executed, referring to the added global variables as required (lines 8–9).

Constant Transformation. Constant values are often served as a signature to fingerprint a particular algorithm or implementation [56]. Hence, perturbing them can impact the systems that rely on such known signatures.

(P21) *Constant Value Splitting*: A constant value N can be converted into two constant values x and y which satisfy $x+y=N$. For example, '`i32.const 16`' is transformed into '`i32.const 10; i32.const 6; i32.add`.' The result of the 'add' is the original constant value '16' and it is stored on the stack.

(P22) *Constant Value Transformation*: For an instruction in a function's body that uses a constant value such as '`i32.const N`,' we replace it with a statement using a global variable '`global.get X`' after inserting an entry for `global X` in the global section.

C. Perturbation Placement and Deployment

For each applicable perturbation, SWAMPED aims to allow users to configure how many (*i.e.*, frequency) and where (*i.e.*, distribution) to be perturbed. However, due to a large number of targets (*i.e.*, elements in sections and instructions), configuring (or specifying) the targets individually is not practical. On the other hand, letting the system choose the targets randomly would lead to biased analysis results. To this

end, SWAMPED employs a concept of perturbation distribution and ratio to decide perturbation targets systematically. By choosing different distributions and ratios, SWAMPED can simulate various perturbation deployment strategies (*e.g.*, applying perturbations more or less on certain areas) in the binary.

1) Perturbation Distribution: Perturbation distribution is designed for selecting targets from a set of available elements or instructions. For example, a uniform distribution selects each element or instruction with equal probability, ensuring even coverage across the section or binary. In contrast, a Gaussian distribution favors elements near a central location, selecting them more frequently while gradually reducing the likelihood of choosing elements farther from the center.

In this work, we implement a beta distribution [57] to determine positions for perturbing binary operations. This distribution is controlled by parameters alpha (α) and beta (β), adjustable to customize the perturbation pattern. By varying these parameters, we can influence the likelihood of selecting positions towards the start, middle, or end of the binary sequence, enabling diverse perturbation methods. This approach provides a flexible mechanism to explore the effects of perturbations across the entire binary structure.

2) Perturbation Ratio: Perturbation ratio is defined as the proportion (*i.e.*, frequency) of perturbed instructions relative to the total number of applicable perturbation targets (*i.e.*, instructions or elements). Let C_a be the total number of perturbation targets and C_p be the number of perturbed targets. The perturbation ratio is calculated as C_p/C_a.

For perturbation methods that insert dummy/dead instructions, C_a includes all instructions in the relevant target section. As these insertion methods can add an arbitrary number of instructions, the perturbation ratio can exceed 100% (*e.g.*, inserting 50 instructions into a section with 50 instructions results in a 100% perturbation ratio). We limited our experiments to a maximum perturbation ratio of 100% to establish a reasonable upper bound. Note that even with this limit, SWAMPED still ends up adding a significant number of instructions (*e.g.*, 31,499 added instructions on average).

D. Execution and Interpretation

Given the perturbed samples, SWAMPED runs each analysis tool (*e.g.*, malware detection tool) and obtains the result. As each analysis tool's results have a slightly different format, we normalize the results into *Detected*, *Suspected*, and *Benign*. Specifically, MineSweeper's output *Positive* and *Negative* are mapped to *Detected* and *Benign*, respectively. Minos, WasmGuard, MalConv, and AvastNet return *Malicious* or *Benign* with a confidence score. *Malicious* maps to *Detected*. *Benign* with confidence less than 80%, which is configurable, maps to *Suspected*; otherwise, *Benign*. MinerRay outputs *Unlikely*, *Probable*, or *Certain*. If all functions are *Unlikely* or unmarked, we assign *Benign*. Otherwise, we assign *Detected*.

After SWAMPED runs each perturbed sample, it signals the system with the interpreted result so that it can decide whether to proceed to the next round of the execution or not. SWAMPED aims to identify perturbation conditions (method, distribution,

ratio) that flip the detection result from malicious to benign, revealing detector weaknesses. Once such a condition is found, it halts further perturbation for that sample.

IV. EVALUATION

We demonstrate the effectiveness of SWAMPED by evaluating Wasm malware detectors against perturbations.

Implementation Details. We implement our framework in Python v3.9. For MineSweeper and MinerRay, we use the publicly available source code from the original studies. For Minos, as the original authors do not publicly release the source code, we use a reimplemented version from the previous work [15], which follows the same training data ratio and model architecture described in the original paper. For WasmGuard, we use the trained model shared by the authors. For the general-purpose detectors MalConv and AvastNet, we train both models using WasmGuard's dataset [58], which consists of 12,018 training samples and 3,006 test samples. Our trained models achieve strong baseline performance on the test set: MalConv and AvastNet achieve 99.87% and 99.93% accuracies and 99.83% and 99.93% F1-scores, respectively, confirming their effectiveness for Wasm malware detection.

When applying the perturbations, we randomly select the code lines using positions sampled from a beta distribution (with α and β set to 1) described in Section III-C. This same sampling approach is consistently applied when selecting artifacts (*e.g.*, functions, import/export names, or custom bytes) for insertion from the benign dataset, ensuring uniform randomness across all perturbation methods. The sampling was performed using Numpy's `default_rng()` random generator, specifically via the `rng.beta($\alpha,\beta$)` function.

Wasm Malware Binary Samples. We use two datasets of malicious Wasm binaries, totaling 74 samples[5].

- *Wasm-Cryptojacker*: We obtain 43 unique cryptojacker samples from prior work [8], [9], [46], each of which is detected by Minos, MineSweeper, and MinerRay. The average size of the samples is 80 KB (62~134 KB). We use this dataset to evaluate the robustness of Minos, MineSweeper, and MinerRay against our perturbations.

- *WasmBench-Malware*: We extract 31 Wasm malware samples from the 1,503-sample evaluation dataset of WasmGuard [12]. Since the dataset was constructed by augmenting a small set of binaries using `wasm-mutate` [27], we identify the original samples by matching hash values with the original source [47]. These 31 samples, consisting of Wasm binaries with an average size of 77 KB (26 KB~122 KB), are used to evaluate the robustness of WasmGuard, MalConv, and AvastNet against our perturbations.

A. Experimental Setup

Creating Variants via Perturbation. We apply SWAMPED to our 74 cryptojacker sample binaries to create 48,840 variants where each of which has different perturbation methods

---

[5]We compare structural properties (*e.g.*, # of sections, instruction categories) of our 74 malware samples with a baseline of 1,482 benign binaries from top-1M websites (Oct 2023–Mar 2024). We find that our malware samples are representative, as the properties between the two sets are similar.

and ratios. Specifically, from 74 samples, we have applied 22 perturbations with different instruction cases resulting in 48,840 samples. For each of the perturbations, we generate 10 variants covering different ratios ranging from 10 to 100 in increments of 10. To ensure robustness, each ratio is tested across three rounds, leading to the final 48,840 variants. We then evaluated the robustness of the six Wasm detectors (Minos, MineSweeper, MinerRay, WasmGuard, MalConv, and AvastNet) against these perturbed binaries.

TABLE II
CLASSIFICATION OF WASM INSTRUCTIONS BY OPERATION CATEGORY.

Category	Ops
Memory (Data) Access	load, size, get, tee
Numeric Arithmetic	add, sub, mul, div, rem
Bitwise Operation	shift, rotate, and, or, xor, clz, ctz, popcnt
Comparison	eq, eqz, ne, lt, gt, le, ge
Integer/FP* Conversion	wrap, extend, trunc, convert
FP* Reinterpretation	promote, demote, reinterpret
FP* Utility	abs, neg, ceil, floor, nearest, sqrt, min, max, copysign

* FP: Floating Point.

Stack Operations. Given the large number of stack-based instructions, we systematically classify Wasm opcodes into six semantic groups, as shown in Table II. Due to the space, it includes only representative opcodes (*e.g.*, div), excluding variants that arise from operand data types (*e.g.*, div_s). Perturbation experiments are performed separately for each category to evaluate detector robustness in a fine-grained manner. For each insertion, a specific instruction and its applicable operands are selected at random. This design allows for controlled and interpretable evaluation across a comprehensive opcode space, rather than focusing only on a small subset such as cryptographic instructions.

Measuring Impact of Perturbations. SWAMPED runs detectors on perturbed samples and checks if their detection decisions differ from those on the original samples. Specifically, we follow the interpretation rules defined in Section III-D. For each pair of a detector and sample, we consider a perturbation to be effective if a decision of a sample is changed from *Detected* or *Suspected* to *Benign*.

The detection timeout for both original and perturbed samples is set to 1 hour. We follow the common practice in the community [59], testing each perturbation 3 times for validity.

B. Perturbation Impacts on Wasm Malware Detectors

We evaluate the impact of our perturbation methods on the decisions (with varying perturbation ratios) of the detectors.[6]

Structural Perturbation. Figure 3 shows how many samples are successfully evaded by the structural perturbations (P1 to P10). For MineSweeper and MinerRay, none of the perturbation methods were effective, because they analyze only the function code section and ignore structural metadata outside the code.

[6]Statistical summaries (average, std, median, and IQR) for the impact of our perturbations are available on https://github.com/SKKU-SecLab/SWAMPED.

Minos's detection is susceptible to perturbations targeting the Function (P3, P4), Data (P8, P9), and Custom (P10) sections, suggesting that Minos may assign significant visual salience to these sections in its grayscale image representation, focusing on them in its classification decisions.

Figure 3 (b) shows the perturbation ratio required to evade the detection of the evaded samples reported in Figure 3 (a). Function Signature Insertion (P1) leads to the evasion on MalConv for 15 (out of 31) samples. The average ratio of 41.33% (±30.91%) means that it required around half of the functions to be cloned. Since the Type section is located toward the beginning of the binary, the inserted entries are likely to fall within the receptive field of MalConv's filters, disrupting its learned byte-level patterns. In contrast, AvastNet is evaded by Data Encryption (P9) perturbation in all samples at a perturbation ratio of 54.84% (±20.11%). The high-entropy transformation introduced by data encryption may significantly alter the statistical characteristics of the Data section, affecting AvastNet's internal embedding distribution. WasmGuard demonstrates robustness against structural perturbations, which may be attributed to its adversarial training strategy. The smallest samples (26–38 KB) show consistently low evasion ratios (*i.e.*, <10%), which may be because fixed-size perturbations are too large relative to the sample size.

Code Perturbation (Instruction Insertion). Figure 4 shows the effective code perturbation methods among those inserting additional instructions (P11~P14), for each detector.

For MineSweeper, Stack Operation Insertion (P12), which inserts bitwise operations, is the only one that is effective. This reflects the MineSweeper's design, which focuses on the frequency of cryptographic instructions profiled in its fingerprint. MinerRay is highly susceptible (30~36 samples out of 43 are evaded by the perturbations) by the NOP Insertion (P11) and various Stack Operation Insertions (P12; including numeric, conversion, reinterpretation, and floating-point instructions). This suggests that while MinerRay leverages semantic-aware analysis, which should not be affected by semantic-preserving perturbations, our perturbations (*i.e.*, instruction insertion) may have impacted their analysis. Besides, Proxy Function Insertion (P14) and Opaque Predicate Insertion (P13) result in the evasion of 5 and 2 samples, respectively.

For all four CNN-based detectors (Minos, WasmGuard, MalConv, and AvastNet), the Stack Operation Insertion (P12) was effective in evading the detection. In particular, Minos, WasmGuard, AvastNet are evaded in every sample under every P12 variant. Regarding the perturbation ratios required to evade, as shown in Figure 4 (b), AvastNet requires comparatively higher perturbation ratios (40%±7.75%~78.28%±16.35%) before evasion occurs. MalConv and WasmGuard are bypassed at slightly lower (10.47%±2.13%~53.33%±24.44%).

Notably, in contrast to WasmGuard and MalConv, Minos and AvastNet remain robust against NOP Insertion (P11) with only 4 and 2 samples evaded despite high perturbation ratios of 95% (±5.77%) and 70% (±14.14%), respectively. This robustness stems from the low-entropy nature of the nop byte (*i.e.*, 0x00). In Minos, such low-contrast pixels are almost
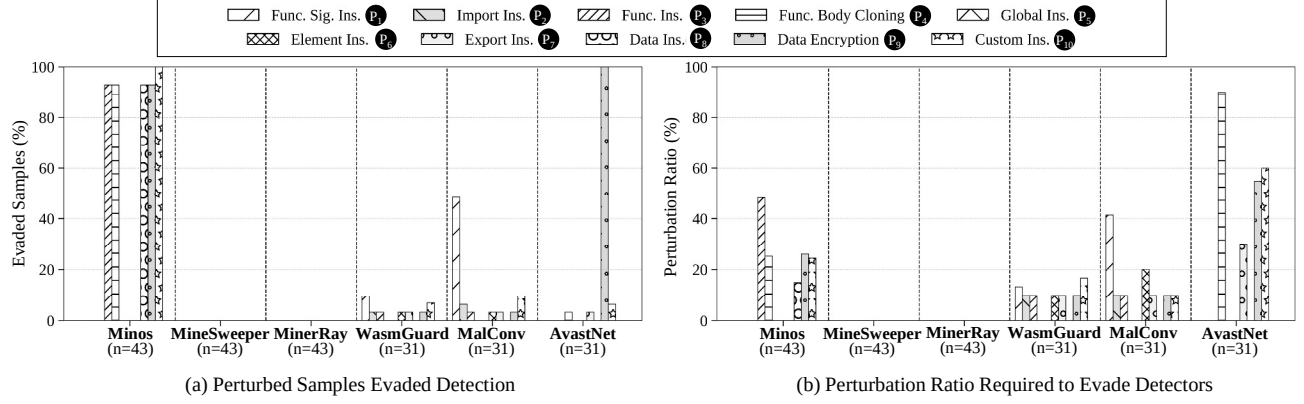
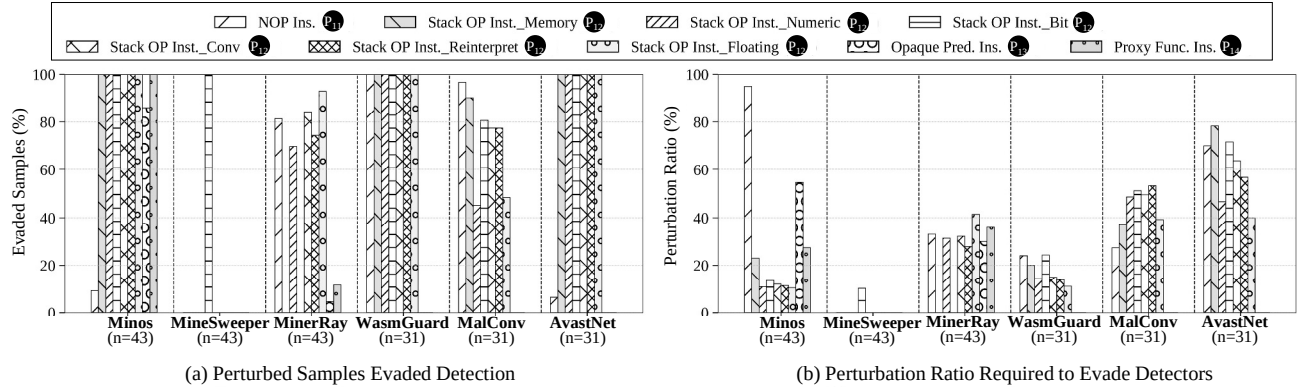Fig. 3. Effects of Structural Perturbations.

Fig. 4. Effects of Code Perturbations involving instruction insertion.

invisible in the grayscale image and therefore leave the learned visual features intact. In AvastNet, down-sampling and global average pooling can dilute the influence of distributed `0x00` bytes. As a result, Minos and AvastNet may ignore them.

Code Perturbation (Instruction/Value Transformation). Figure 5 shows code transformation perturbations (P_{15}~P_{20}) and value transformation perturbations (P_{21}, P_{22}) effectively impacting the detectors. For MineSweeper, as it relies on the frequency of cryptographic instructions (*i.e.*, shl, shr, or, and xor), it is highly susceptible to the transformation of those instructions (P_{17} and P_{20}), with less than 20% of the perturbations as shown in Figure 5 (b).

Interestingly, MinerRay is susceptible to every instruction and value transformation perturbation, with the most pronounced evasions–74.42%~95% of samples misclassified– arising from Instruction Substitution (P_{16}~P_{19}) and Direct Call to Indirect Call Transformation (P_{15}), suggesting that MinerRay's semantic-aware analysis may not handle diverse instructions/semantics as well as indirect calls. Another noticeable trend is that MalConv and AvastNet are not susceptible to any code transformation perturbations (P_{16}~P_{20}). We further investigate the reason behind it and find that all the code transformation perturbations are applied to small areas of the binary (less than 7% on average). This means that the impact of those perturbations might be insufficient to affect the

model's performance. However, we observe that Minos and WasmGuard are susceptible to certain code transformations, which may suggest that their models have a greater focus on specific instructions (*e.g.*, sub, add, shl, shr, or, and xor), which might be caused by their fine-tuning process for Wasm malware samples, such as cryptojackers. Value transformation perturbations (P_{21}, P_{22}) are often effective, as we find that such transformation can impact up to 29% of the binary.

C. Insights from the Experiment Results

We make a few observations based on our evaluation results. First, fine-tuned and specialized detectors (MineSweeper and MinerRay) are less susceptible to non-code perturbations while overly sensitive to perturbations on certain instructions they focus on. As a result, those are easy to evade if perturbing specific instructions. Second, CNN-based detectors (Minos, WasmGuard, MalConv, and AvastNet) are susceptible to the perturbations that affect the overall layout (*e.g.*, the distribution of instructions or data). In general, as long as the binary's data distribution is altered, it is often affected. Third, there exists a level of specialization among CNN-based techniques. While WasmGuard does not include code or data targeting specific instructions in its adversarial training, it is substantially affected by shift and bitwise instructions. This might be caused by either overfitting [60] or shortcut feature [61] behaviors.
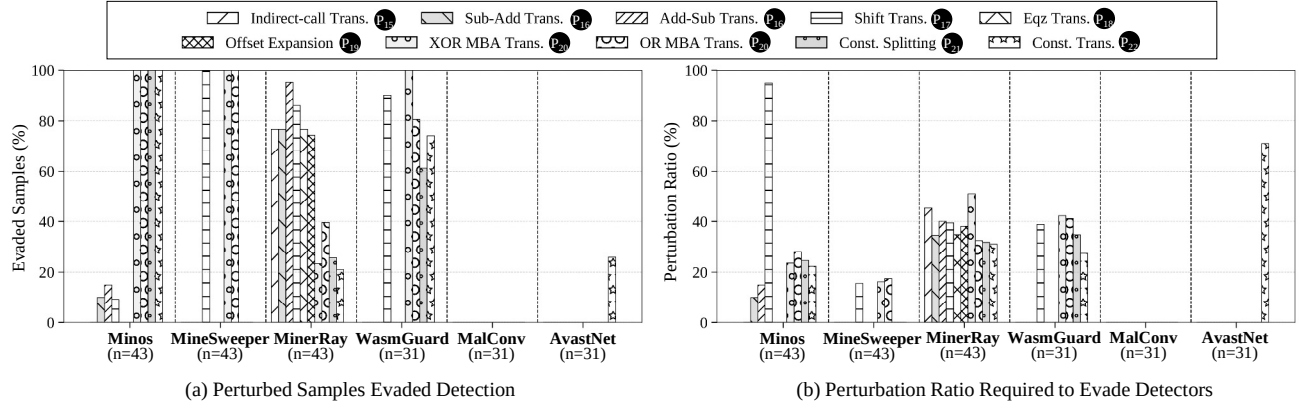
Fig. 5. Effects of Code Perturbations involving instruction and constant transformations.

(a) Perturbed Samples Evaded Detection

(b) Perturbation Ratio Required to Evade Detectors

Fourth, we observe that while the performance of those detectors in detecting malware is relatively stable and similar, their reactions to the perturbed samples are more diverse. This is even true for relatively similar techniques, such as MalConv and AvastNet, meaning that under an adversary capable of perturbations, their real-world performance would vary significantly. This echoes this paper's motivation: understanding the resilience of detectors against perturbation is crucial.

D. Overhead

We evaluate SWAMPED's overhead in two aspects: *i) Perturbation Overhead*: the time and memory required to generate perturbed binaries, and *ii) Execution Overhead*: the slowdown when executing perturbed binaries relative to their originals. All experiments were conducted on Ubuntu 18.04.5 equipped with an Intel Xeon Gold 6230 (2.10GHz) and 566 GB RAM. **Perturbation Overhead.** Across all perturbation methods and perturbation ratios, the overhead remains stable and primarily scales with binary size. For example, perturbing a 62 KB binary took 1.32s with 120 MB memory, while a 131 KB binary required 2.04s and 140 MB memory. Overall, SWAMPED perturbed most samples within seconds using modest memory. **Execution Overhead.** We instantiated each of the 74 malware samples in Node.js 22.19.0 (V8 12.4.254), disabling tier-up and lazy compilation to enforce baseline Liftoff JIT execution. As the samples lack JavaScript artifacts, we supplied minimal runtime resources (memory, table, and globals) and replaced import functions with no-op stubs. To measure execution time, each exported function was first executed 1,000 times for warm-up, and we took the median of the next 100 runs.

Table III shows the median ΔCPU time and relative overhead for each perturbation method. For most perturbations, the overhead is statistically negligible. P_{11}, P_{12}, P_{17}, P_{19}, and P_{21} yield slight speed ups, which we attribute to measurement noise at the nanosecond scale rather than genuine performance gains. Notably, while P_{12} inserts a large number of stack operations, enlarging binary sizes by almost threefold, the inserted operations are optimized by the V8 engine at runtime, leading to virtually no overhead. P_{14} and P_{16} (Instruction sub-

stitution and lightweight control-flow change) incur negligible slowdowns, with median overheads of a few nanoseconds. In contrast, P_{13}, P_{15}, P_{20}, and P_{22} showed more noticeable slowdowns, ranging from several tens to a few hundred nanoseconds, due to added control-flow checks, expanded bit-wise expressions, indirect calls, or global constant accesses. To assess statistical significance, we performed a Mann-Whitney U test [62], which revealed that only P_{13} and P_{20} produced statistically significant differences ($p < 0.05$), while the other perturbations did not deviate significantly from the baseline.

TABLE III
EXECUTION OVERHEAD OF CODE PERTURBATIONS ON 74 SAMPLES.

ID	Perturbation Method	ΔCPU Time (ns)*		Overhead†
		Median	IQR	
P_{20}	XOR/OR to MBA	225.86	253.82	98%
P_{13}	Opaque Predicate Insertion	102.54	143.39	73%
P_{22}	Constant Value Transformation	53.05	20.11	16%
P_{15}	Direct → Indirect Call	43.75	51.98	15%
P_{18}	eqz Transformation	2.50	20.12	19%
P_{16}	Add/Sub Transformation	2.00	10.79	5%
P_{14}	Proxy Function Insertion	1.50	9.98	7%

* (perturbed execution time − original execution time) (ns).
† (perturbed execution time / original execution time − 1) * 100 (%).

V. DISCUSSION

Threats to Validity. For *external validity*, our evaluation set is dominated by cryptojacker samples (43 out of 74), reflecting trends in Wasm malware research. While this may bias the results, the inclusion of 31 non-cryptojacker samples and general-purpose detectors broadens the applicability of our findings beyond cryptojacking. For *construct validity*, SWAMPED applies semantics-preserving perturbations without prioritizing semantically critical regions. This may not fully reflect real-world adversaries who often target logic-intensive code, potentially reducing the effectiveness of the perturbations. For *internal validity*, our dataset may contain hidden correlations (*e.g.*, compiler toolchains, obfuscation patterns) that could confound the relationship between perturbations and detector failures. While we deduplicated identical binaries,

such correlations are difficult to fully eliminate and may still influence the results.

Uniqueness of Wasm Perturbation. Wasm and its binary have unique constraints compared to other binaries (*e.g.*, x86), which distinguish Wasm perturbations and their impact. First, Wasm's underlying stack-based virtual machine results in a unique focus on stack operations. In particular, due to its versatile stack operations, insertion and manipulation of the stack operations are both feasible and effective, as shown in our evaluation. Second, Wasm's structured control flows and validation rules prohibit perturbations relying on arbitrary jumps [19], self-modifying code [20], and instruction overlapping [21], [22]. Also, control-flow obfuscations effective in other binaries become less impactful (*e.g.*, P₁₃ affects only Minos in our evaluation). Third, Wasm's strong inter-section dependencies affect implementations of existing perturbations and their impact. Specifically, a perturbation in one section often requires coordinated changes in others (*e.g.*, P₁₅ spans `Function-Table-Element`; P₂₀ and P₂₂ touch `Function-Global`). For instance, P₂₂ modifies both instructions and the `Global` section, whereas its x86 counterpart would only alter the memory location of a `mov` instruction. We further observe that multi-section perturbations often yield more evasive effects than single-section ones (*e.g.*, P₂₁ vs. P₂₂).

Extensibility and Modularity. While we focus on Wasm malware detectors due to the need for understanding real-world malware evasion, SWAMPED's semantics-preserving perturbations apply to any Wasm binary analyzer, making it a general robustness testing framework. To demonstrate this versatility, we evaluated SWAMPED against VetEOS [63], a Wasm smart contract vulnerability detector. All 24 vulnerable binaries shipped with VetEOS became undetected after applying NOP Insertion and Stack Operation Insertion with only 10% perturbation ratio, revealing significant brittleness in the analyzer. SWAMPED uses a modular design with separate components for parsing, perturbation, and output interpretation. New transformations require minimal code changes, and transformation rules can be added easily. This design supports robustness evaluation across diverse Wasm analyzers.

Resilience of Perturbations. Some of our perturbations can be reversed or eliminated—particularly insertion-based ones (*e.g.*, NOP, Stack Operation, Opaque Predicate)—though it is often challenging to determine whether a given instruction is inserted or not in the first place. In addition, some perturbations creating one-to-many transformation relationships are challenging to reverse. MBA and Shift-transformations are such examples, as multiple original forms can be obfuscated into the same form. Defenses trained on specific perturbations (*e.g.*, WasmGuard) gain robustness for those seen in training, but have limited or negative impact on other perturbations. For example, retraining MalConv with 1,000 Stack Operation Insertion samples improved detection for the samples while weakening it against Shift Operation Transformation, suggesting the risk of local gains at the cost of broader resilience.

VI. RELATED WORK

C Source- and LLVM-Level Perturbations. Several studies have explored evasion techniques for Wasm malware detectors at C source-level and LLVM-level. Bhansali *et al.* [41] applied Tigress obfuscations [64] and CROW [42] introduced an LLVM-based diversification tool that substitutes instruction sequences. Harnes *et al.* [40] conducted a large-scale study of multi-level obfuscation using Tigress and emcc-obf. In contrast, we directly perturb Wasm binaries, independent of source languages and compilers, adapting and extending prior transformations into binary-level forms (*e.g.*, Function Body Cloning, Add/Sub Operation Transformation).

Wasm Binary-Level Perturbations. In addition to source-level and LLVM-level approaches, recent work has explored Wasm binary-level perturbations. Javier *et al.* [15] used `wasm-mutate` [27] to build a binary diversification pipeline including peephole mutations, structural changes, and control-flow edits. While peephole mutation was most effective against Minos and VirusTotal, it applied random, coarse-grained rules (*e.g.*, identity, commutativity), requiring hundreds of attempts per sample or previously applied strategies being nullified by subsequent ones. Cao *et al.* [28] introduced WASMixer, an obfuscator applying direct call transformation, opaque predicates, and memory encryption to evade analysis tools. Madvex [65] applied adversarial gadgets like `const-drop` sequences and opaque loops to Wasm binaries. Chmiel *et al.* [66] conceptually discussed evading MineSweeper by breaking loop analysis using `nop` insertion, though our experiments show this technique alone is ineffective. Unlike prior work, we construct a comprehensive and fine-grained set of Wasm binary perturbations—including both structural and code-level transformations, enabling systematic, controlled experiments. Our large-scale evaluation not only integrates and refines previous methods but also quantifies how specific perturbations affect detector performance under varying perturbation ratios.

VII. CONCLUSION

This work presents SWAMPED, a framework for systematically evaluating Wasm malware detector robustness through 22 semantics-preserving perturbation methods targeting section entries and function code. Using SWAMPED, we evaluate six detectors: four Wasm-specific models (Minos, MineSweeper, MinerRay, WasmGuard) and two bytecode classifiers (MalConv, AvastNet). Our experiments on 48,840 perturbed variants reveal systematic evasion: stack operation insertions disrupt DNN-based models, instruction frequency changes mislead profiling-based methods, and control-flow modifications break rule-based analyzers, highlighting fundamental weaknesses in current Wasm malware detection.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[2] L.-N. Pouchet and T. Yuki, "PolyBenchC-4.2.1," 2016. [Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.git

[3] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the Performance of WebAssembly Applications," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2021.

[4] A. Lonkar and S. Chandrayan, "The Dark Side of WebAssembly," 2018. [Online]. Available: https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly

[5] M. Maganu, "WebAssembly is Abused by eCriminals to Hide Malware," 2021. [Online]. Available: https://www.crowdstrike.com/en-us/blog/ecriminals-increasingly-use-webassembly-to-hide-malware

[6] J. Whitehorn, "JsMiner," 2011. [Online]. Available: https://github.com/jwhitehorn/jsMiner

[7] F. N. Naseem, A. Aris, L. Babun, E. Tekiner, and A. S. Uluagac, "MINOS: A Lightweight Real-Time Cryptojacking Detection System," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[8] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[9] A. Romano, Y. Zheng, and W. Wang, "MinerRay: Semantics-Aware Analysis for Ever-Evolving Cryptojacking Detection," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[10] C. Komiya, N. Yanai, K. Yamashita, and S. Okamura, "JABBERWOCK: A Tool for WebAssembly Dataset Generation towards Malicious Website Detection," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023.

[11] W. Bian, W. Meng, and Y. Wang, "Poster: Detecting WebAssembly-Based Cryptocurrency Mining," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[12] Y. Sun, H. Chen, Z. Fu, W. Lv, Z. Liu, and H. Liu, "WasmGuard: Enhancing Web Security through Robust Raw-Binary Detection of WebAssembly Malware," in *Proceedings of the ACM on Web Conference (WWW)*, 2025.

[13] Y. Xia, P. He, X. Zhang, P. Liu, S. Ji, and W. Wang, "Static Semantics Reconstruction for Enhancing JavaScript-WebAssembly Multilingual Malware Detection," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2023.

[14] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2021.

[15] J. Cabrera-Arteaga, M. Monperrus, T. Toady, and B. Baudry, "WebAssembly Diversification for Malware Evasion," *Computers & Security*, 2023.

[16] L. Demetrio, S. E. Coull, B. Biggio, G. Lagorio, A. Armando, and F. Roli, "Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection," *ACM Transactions on Privacy and Security*, 2021.

[17] B. Jin, J. Choi, J. B. Hong, and H. Kim, "On the Effectiveness of Perturbations in Generating Evasive Malware Variants," *IEEE Access*, 2023.

[18] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *Proceedings of the USENIX Security Symposium (Security)*, 2020.

[19] V. Balachandran and S. Emmanuel, "Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 4, 2013.

[20] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2003.

[21] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2006.

[22] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: a hidden code extractor for packed executables," in *Proceedings of the ACM Workshop on Recurring Malcode (WoRM)*, 2007.

[23] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," *arXiv:1710.09435*, 2017.

[24] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[25] E. Wen, "Browserify: Empowering Consistent and Efficient Application Deployment across Heterogeneous Mobile Devices," PhD Thesis, The University of Auckland, 2020.

[26] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark, "A First Look at Browser-Based Cryptojacking," in *Proceedings of the IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2018.

[27] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "Wasm-Mutate: Fast and Effective Binary Diversification for WebAssembly," *Computers & Security*, 2024.

[28] S. Cao, N. He, Y. Guo, and H. Wang, "WASMixer: Binary Obfuscation for WebAssembly," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2024.

[29] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild," in *Proceedings of the ACM Web Conference (WWW)*, 2019.

[30] W. Bian, W. Meng, and M. Zhang, "MineThrottle: Defending against Wasm In-Browser Cryptojacking," in *Proceedings of the ACM Web Conference (WWW)*, 2020.

[31] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy," in *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2020.

[32] F. Tommasi, C. Catalano, U. Corvaglia, and I. Taurino, "MinerAlert: An Hybrid Approach for Web Mining Detection," *Journal of Computer Virology and Hacking Techniques*, 2021.

[33] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2018.

[34] T. Sermchaiwong and J. Shen, "Dynamic Graph-based Fingerprinting of In-browser Cryptomining," *arXiv:2505.02493*, 2025.

[35] I. Petrov, L. Invernizzi, and E. Bursztein, "CoinPolice: Detecting Hidden Cryptojacking Attacks with Neural Networks," *arXiv:2006.10861*, 2020.

[36] O. Suciu, S. E. Coull, and J. Johns, "Exploring Adversarial Examples in Malware Detection," in *Proceedings of the IEEE Security and Privacy Workshops (SPW)*, 2019.

[37] Y. Sun, H. Chen, J. Guo, A. Sun, Z. Li, and H. Liu, "RoMA: Robust Malware Attribution via Byte-level Adversarial Training with Global Perturbations and Adversarial Consistency Regularization," *arXiv:2502.07492*, 2025.

[38] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv:1412.6572*, 2014.

[39] W. C. Group, "WebAssembly Binary Toolkit," 2024. [Online]. Available: https://github.com/WebAssembly/wabt.git

[40] H. Harnes and D. Morrison, "Cryptic Bytes: WebAssembly Obfuscation for Evading Cryptojacking Detection," *arXiv:2403.15197*, 2024.

[41] S. Bhansali, A. Aris, A. Acar, H. Oz, and A. S. Uluagac, "A First Look at Code Obfuscation for Webassembly," in *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2022.

[42] J. Cabrera-Arteaga, O. Floros, B. Baudry, and M. Monperrus, "CROW: Code Diversification for WebAssembly," in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2021.

[43] D. Gibert, M. Fredrikson, C. Mateu, J. Planes, and Q. Le, "Enhancing the Insertion of NOP Instructions to Obfuscate Malware via Deep Reinforcement Learning," *Computers and Security*, 2022.

[44] L. Zhang, P. Liu, Y.-H. Choi, and P. Chen, "Semantics-Preserving Reinforcement Learning Attack Against Graph Neural Networks for Malware Detection," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[45] K. Lucas, S. Pai, W. Lin, L. Bauer, M. K. Reiter, and M. Sharif, "Adversarial Training for Raw-Binary Malware Classifiers," in *Proceedings of the USENIX Security Symposium (Security)*, 2023.

[46] A. Romano and W. Wang, "WASim: Understanding WebAssembly Applications through Classification," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020.

[47] A. Hilbig, D. Lehmann, and M. Pradel, "WasmBench," 2020. [Online]. Available: https://github.com/sola-st/WasmBench

[48] B. Sheridan and M. Sherr, "On Manufacturing Resilient Opaque Constructs Against Static Analysis," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2016.

[49] L. Zobernig, S. D. Galbraith, and G. Russello, "When are Opaque Predicates Useful?" in *Proceedings of the IEEE International Conference On Trust, Security And Privacy In Computing And Communications/IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, 2019.

[50] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[51] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?" *ACM Computing Surveys*, 2016.

[52] Ş. Andrei and C. Masalagiu, "About the Collatz Conjecture," *Acta Informatica*, 1998.

[53] J. Lee and W. Lee, "Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[54] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, "MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation," in *Proceedings of the USENIX Security Symposium (Security)*, 2021.

[55] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi, "Loki: Hardening Code Obfuscation Against Automated Attacks," in *Proceedings of the USENIX Security Symposium (Security)*, 2022.

[56] P. Lestringant, F. Guihéry, and P.-A. Fouque, "Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.

[57] A. K. Gupta and S. Nadarajah, *Handbook of Beta Distribution and Its Applications*. CRC Press, 2004.

[58] Yuxia-Sun, "WasmMal," 2025. [Online]. Available: https://github.com/Yuxia-Sun/WasmMal

[59] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[60] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding Deep Learning (Still) Requires Rethinking Generalization," *Communications of the ACM*, 2021.

[61] R. Geirhos, J.-H. Jacobsen, C. Michaelis, R. Zemel, W. Brendel, M. Bethge, and F. A. Wichmann, "Shortcut Learning in Deep Neural Networks," *Nature Machine Intelligence*, 2020.

[62] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric Statistical Methods*, 3rd ed., 2015.

[63] L. T. Li, N. He, H. Wang, and M. Zhang, "VETEOS: Statically Vetting EOSIO Contracts for the "Groundhog Day" Vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2024.

[64] C. Collberg, "Tigress: A Diversifying Virtualizer/Obfuscator for C," 2011. [Online]. Available: https://tigress.wtf

[65] N. Loose, F. Mächtle, C. Pott, V. Bezsmertnyi, and T. Eisenbarth, "Madvex: Instrumentation-Based Adversarial Attacks on Machine Learning Malware Detection," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2023.

[66] K. Chmiel and P. Rajba, "How to Evade Modern Web Cryptojacking Detection Tools? A Review of Practical Findings," in *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*, 2024.