WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly

Javier Cabrera-Arteaga[a,*], Nick Fitzgerald,[b], Martin Monperrus[a] and Benoit Baudry[a]

[a]KTH Royal Institute of Technology, Stockholm, Sweden
[b]Fastly Inc., San Francisco, USA

ABSTRACT

WebAssembly has quickly ascended as a promise for web development, renowned for its efficiency and expanding utility beyond browser environments. Yet, the burgeoning ecosystem of WebAssembly compilers and tools has created a need for robust software diversification techniques. Traditional approaches, often tied to specific compilation pipelines, can be restrictive in their scope and application.

We introduce WASM-MUTATE, a compiler-agnostic WebAssembly diversification engine. It is engineered to fulfill key criteria: the rapid generation of semantically equivalent yet behaviorally diverse WebAssembly variants, universal compatibility with existing WebAssembly programs, and the capability to counter high-risk security threats. Utilizing an e-graph data structure, WASM-MUTATE pioneers its usage in the realm of software diversification. Our assessments reveal that WASM-MUTATE can efficiently generate tens of thousands of unique WebAssembly variants in a matter of minutes. Additionally, it can produce variants with distinct machine code instruction traces and memory profiles in milliseconds. Notably, WASM-MUTATE's generated variants are fortified against Spectre attacks. This research not only delivers a potent tool for WebAssembly diversification but also contributes valuable insights for the development of more secure and resilient WebAssembly applications.

1. Introduction

WebAssembly is the fourth official language of the web, complementing HTML, CSS and JavaScript as a fast, platform-independent binary format [21, 40]. Since its introduction in 2015, it has seen rapid adoption, with support from all major browsers, including Firefox, Safari and Chrome. WebAssembly has also been adopted outside of browsers, with world-leading execution platforms like Fastly using it as a foundational technology for their content delivery network [17]. In addition to major ones like LLVM, more and more compilers and tools can output WebAssembly binaries [23, 45, 26]. With this prevalence, it is of utmost importance to design software protection techniques for WebAssembly [27].

Software diversification is a well-known software protection technique [12, 4, 19], consisting of producing numerous variants of an original program, each retaining equivalent functionality. Software diversification in WebAssembly has many important application domains, such as optimization [5] and malware evasion [8]. It can also been used for fuzzing, a salient example of this was the discovery of a CVE in Fastly in 2021 [18], achieved through automated transformations to a WebAssembly binary.

To develop an effective WebAssembly diversification engine, several key requirements must be met. First, the engine should be language-agnostic, enabling diversification of any WebAssembly code, regardless or the source programming language and compiler toolchain. Second, it must have the capability to swiftly generate semantically equivalent variants of the original code. The speed at which this diversification occurs holds potential for real-time applications, including moving target defense [6]. The engine should also possess the ability to counter attackers by producing sufficiently distinct code variants. This paper present an original system, WASM-MUTATE, that addresses all these requirements.

WASM-MUTATE is a tool to automatically transforms a WebAssembly binary program into a variant binary program that preserves the original functionality. The core of the diversification engine relies on an e-graph data structure [48]. To the best of our knowledge, this work is the first to use an e-graph for software diversification in WebAssembly. An e-graph offers one essential property for diversification: every path through the e-graph represents a functionally equivalent variant of the input program [48, 37]. A random e-graph traversal can also be very efficient, supporting the generation of tens of thousands of equivalent variants from a single seed program in minutes [29] Consequently, the choice of e-graphs is the key to build a diversification tool that is both effective and fast. We have designed 135 rewriting rules in WASM-MUTATE, which can transform the e-graph from fine to coarse grained levels.

We assess the effectiveness of WASM-MUTATE with respect to its capacity at generating variants, which code is different from the original and which execution exhibit diverse instruction and memory traces. Our empirical evaluation reuses an existing corpus from the diversification literature [7]. We also measure the speed at which WASM-MUTATE is able to generate the first variant that exhibits a trace different from the original. Our security security assessment of WASM-MUTATE consists in evaluating the degree to which diversification can mitigate Spectre attacks. This assessment

*Corresponding authors
javierca@kth.se (J. Cabrera-Arteaga); toady@eecs.kth.se (N. Fitzgerald,); monperrus@kth.se (M. Monperrus); baudry@kth.se (B. Baudry)
ORCID(s): 0000-0001-9399-8647 (J. Cabrera-Arteaga); 0000-0002-0209-2805 (N. Fitzgerald,); 0000-0003-3505-3383 (M. Monperrus); 0000-0002-4015-4640 (B. Baudry)

is made with WebAssembly programs that have been previously identified as vulnerable to Spectre attacks [36].

Our results demonstrate that WASM-MUTATE can generate thousands of variants in minutes. These variants have unique machine code after compilation with cranelift (static diversity) and the variants exhibit different traces at runtime (dynamic diversity). Our experiments also provide evidence that the generated variants are hardened against Spectre attacks. To sum up, the contributions of this work are:

- The design and implementation of a WebAssembly diversification pipeline, based on semantic-preserving binary rewriting rules.
- Empirical evidence on the diversity of variants created by WASM-MUTATE, both in terms of static binaries and execution traces.
- Demonstration that WASM-MUTATE can protect WebAssembly binaries against timing side-channel attacks, specifically, Spectre.
- An open-source repository, where WASM-MUTATE is publicly available for future research https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate.

This paper is structured as follows. In section 2, we introduce WebAssembly, the concepts of semantic equivalence and what we state as a rewriting rule. In section 3, we explain and detail the architecture and implementation of WASM-MUTATE. We formulate our research questions in section 4, answering them in section 5. We discuss open challenges related to our research in section 6, in order to help future research projects on similar topics. In section 7 we highlight works related to our research on software diversification. We finalize with our conclusions section 8.

2. Background

In this section, we define and formulate the foundation of this work: WebAssembly and its runtime structure, semantic equivalence modulo input, rewriting rules and e-graphs. Along with the paper, we use the terms, metrics and concepts defined here.

2.1. WebAssembly

WebAssembly (Wasm) is a binary instruction set initially meant for the web, and now also used in the backend. It was adopted as a standardized language by the W3C in 2017, building upon the work of Haas et al. [21]. One of Wasm's primary advantages is that it defines its own Instruction Set Architecture (ISA), which is both platform-independent. As a result, a Wasm binary can execute on virtually any platform, including web browsers and server-side environments. WebAssembly programs are compiled ahead-of-time from source languages such as C/C++, Rust, and Go, utilizing compilation pipelines like LLVM.

```rust
fn main() {
    let mut arr = [1, 2, 3, 4, 5];
    // Variable assignment
    let mut sum = 0;
    // Loop and memory access
    for i in 0..arr.len() {
        sum += arr[i];
    }
    // Use of external function
    println!("Sum of array elements: {}", sum);
}
```

Listing 1: Rust program containing function declaration, loop, conditional and memory access.

```
(module
  (@custom "producer" "llvm.." )
  (import "env" "println" (func $println (param i32)))
  (memory 1)
  (export "memory" (memory 0))
  (func $main
    (local $sum i32)
    (local $i i32)
    (local $arr_offset i32)
    ; Initialize sum to 0 ;
    i32.const 0
    local.set $sum
    ; Initialize arr_offset to point to start of the array
        in memory ;
    i32.const 0
    local.set $arr_offset
    ; Initialize the array in memory;
    i32.const 0
    i32.const 1
    i32.store
    ...
    i32.store
    ...
    loop
      local.get $i
      i32.const 5
      i32.lt_s
      if
        ; Load array[i] and add to sum ;
        local.get $arr_offset
        local.get $i
        ...
        ; Increment i ;
        local.get $i
        i32.const 1
        i32.add
        local.set $i
        br 0
      else
        ; End loop ;
        i32.const 0
      end
    end

    ; Call external function to print sum ;
    local.get $sum
    call $println
  )
  ; Start the main function ;
  (start $main)
  )
```

Listing 2: Simplified WebAssembly code for Rust code in Listing 1.

WebAssembly programs operate on a virtual stack that allows primitive data types. Additionally, a WebAssembly

program might include several custom sections. For example, binary producers such as compilers use custom sections to store metadata, such as the name of the compiler that generates the Wasm code. A WebAssembly program also declares memory sections and globals, which are used to store, manipulate and share data during program execution, e.g. to share data with the host engine of the WebAssembly binary.

WebAssembly is designed with isolation as a primary consideration. For instance, a WebAssembly binary cannot access the memory of other binaries or cannnot interact directly with browser's APIs, such as the DOM or the network. Instead, communication with these features is constrained to functions imported from the host engine, ensuring a secure and safe Wasm environment. Moreover, control flow in WebAssembly is managed through explicit labels and well-defined blocks, which means that jumps in the program can only occur inside blocks, unlike regular assembly code [22]. In Listing 1, we provide an example of a Rust program that contains a function declaration, a loop, a loop conditional, and a memory access. When the Rust code is compiled to WebAssembly, it produces the code shown in Listing 2. The stack operations are folded with parentheses. The module in the example contains the components described previously.

The WebAssembly runtime structure is described in the WebAssembly specification and it includes 10 key elements: the Store, Stack, Locals, Module Instances, Function Instances, Table Instances, Memory Instances, Global Instances, Export Instances, and Import Instances. These components interact during the execution of a WebAssembly program, collectively defining the state of a program during its runtime.

Two of these elements, the Stack and Memory instances, are particularly significant in maintaining the state of a WebAssembly program during its execution. The Stack holds both values and control frames, with control frames handling block instructions, loops, and function calls. Meanwhile, Memory Instances represent the linear memory of a WebAssembly program, consisting of a contiguous array of bytes. In this paper, we highlight the aforementioned two components to define, compare and validate the state of two Wasm programs during their execution.

2.2. Semantic Equivalence

Semantic equivalence refers to the notion that two programs or functions are considered equivalent if, for a given specified input domain, they produce the same output values or have the same observable behavior [30]. In other words, the semantics of the two programs are equivalent when the input-output relationship (w/ possibly some abstraction), even if the internal implementation details or the structure of the programs differ.

Let us illustrate this with an example. Assume two programs P and P' (Listing 3 and Listing 4 respectively) where P' is the result of modifying a code in the first instruction of its unique function. The program P' has two extra instructions right before returning from the function. The remaining components of the original binary are not modified.

```
func (;0;) (type 0) (param i32
      f32) (result i64)
    i64.const 1
```

Listing 3: Program P.

```
func (;0;) (type 0) (param i32
      f32) (result i64)
    i64.const 1
    i32.const 42
    i32.drop
```

Listing 4: Program P', transformation of program P.

The state of the program P when entering the function is its stack $[S]$, the program P' has the same state before executing the function. The input values of the function for both programs are L, their outputs are the top of the stack at the end of the execution.

Program P has the state $[[S : i32.const\ 1]]$ just before returning from the function execution. When we trace the states of the program P', we can construct the following sequence of states:

1. $[[S : i32.const\ 1]]$ the integer constant 1 is now on the top of the stack.
2. $[[S : i32.const\ 1, i32.const\ 42]]$ the integer constant 32 is the top of the stack.
3. $[[S : i32.const\ 1]]$ the top of the stack is dropped. The function execution stops.

Notice that, the stack state of program P' is the same as program P. Thus, we can say that these two programs are semantically equivalent. Even though the programs share semantic equivalence, they display differences during execution. Specifically, P' stresses more on the stack by adding and subsequently dropping more values. These subtle yet significant differences form the crux of the diversification approaches discussed in this study.

2.3. Rewriting rule

Our definition of a rewriting rule draws from the one proposed by Sasnauskas et al. [41], and integrates a predicate to specify the replacement condition. Concretely, a rewriting rule is defined as a tuple, denoted as `(LHS, RHS, Cond)`. Here, `LHS` refers to the code segment slated for replacement, `RHS` is the proposed replacement, and `Cond` stipulates the conditions under which the replacement is acceptable. Importantly, `LHS` and `RHS` are meant to be semantically equivalent, per the definition of previous section.

For example, the rewriting rule `(x, x i32.or x, {})` implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, absent any specific conditions. Notice that, for this specific rule, the commutative property shared by `LHS` and `RHS`, symbolized as `(LHS, RHS)=(RHS, LHS)`. Besides, the `Cond` element could be an arbitrary criterion. For instance, the condition for applying the aforementioned rewriting rule could be to ensure that the newly created binary file does not exceed a threshold binary size.

Based on our understanding, our research is the first to apply the concept of rewriting rules to WebAssembly. This

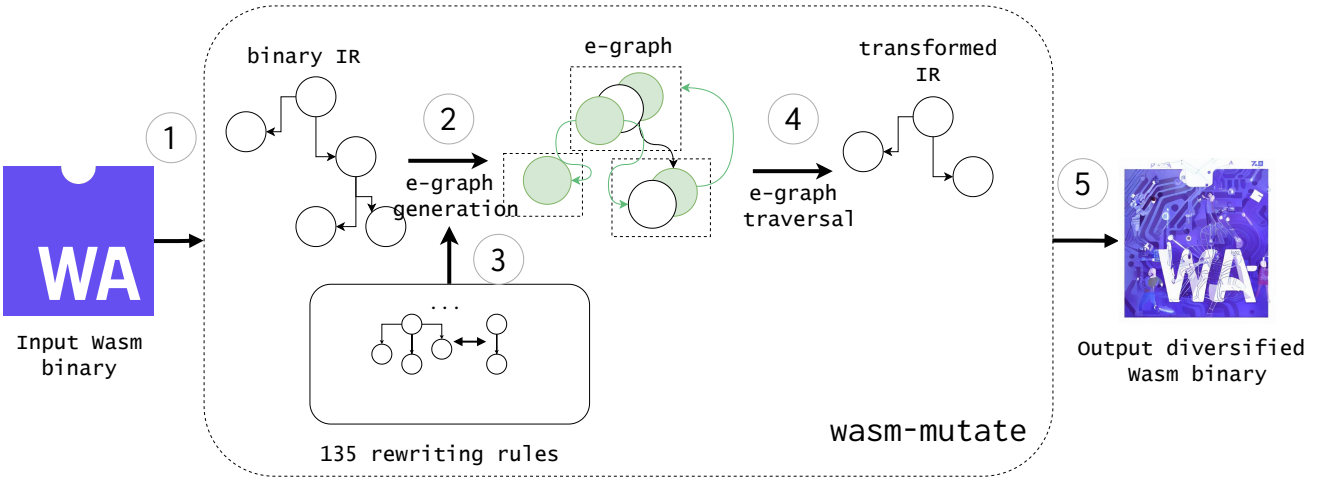

Figure 1: WASM-MUTATE workflow and high level architecture. It generates semantically equivalent variants from a given WebAssembly binary input. Its central approach involves synthesizing these variants by substituting parts of the original binary using rewriting rules, boosted by a diversification space traversals using e-graphs(refer to subsection 3.3)

will expand the potential use cases of wasm-mutate. Beyond its role as a diversification tool, it can also be used as a standard tool for conducting program transformations in WebAssembly.

3. Design of Wasm-Mutate

In this section we present WASM-MUTATE, a tool to diversify WebAssembly binaries and produce semantically equivalent variants.

3.1. Overview

The primary objective of WASM-MUTATE is to perform diversification, i.e., generate semantically equivalent variants from a given WebAssembly binary input. WASM-MUTATE's central approach involves synthesizing these variants by substituting parts of the original binary using rewrite rules. It leverages a comprehensive set of rewrite rules, boosted by a diversification space traversals using e-graphs(refer to subsection 3.3).

In Figure 1 we illustrate the workflow of WASM-MUTATE: it starts with a WebAssembly binary as input (1). It parses the original binary (2), turning the input program into appropriate abstractions, in particular WASM-MUTATE builds the control flow graph and data flow graph. Using the defined rewriting rules, WASM-MUTATE builds an e-graph (3) for the original program. An e-graph packages every possible equivalent code derivable from the given rewriting rules [48, 37]. Thus, at this stage, WASM-MUTATE exploits a key property of e-graphs: any path traversal through the e-graph results in a semantically equivalent code. Then, the diversification process starts, with parts of the original program being randomly replaced by traversal of the e-graph (4). The outcome of WASM-MUTATE is a semantically equivalent variant of the original binary (5). The tool guarantees semantically equivalent variants because each individual rewrite rule is semantic preserving.

3.2. WebAssembly Rewriting rules

In total, there are 135 possible rewriting rules implemented in WASM-MUTATE, those rules are grouped under several categories, called hereafter meta-rules. For example, 125 rewriting rules are implemented as part of a peephole meta-rule. There are 7 meta-rules that we present next.

Add type: In WebAssembly, the type section wraps definitions of signatures for the binary functions. WASM-MUTATE implements two rewrite rules, one of which is illustrated in the following rewriting rule.

```
LHS (module
  (type (;0;) (func (param i32) (result i64)))

RHS (module
  (type (;0;) (func (param i32) (result i64)))
+ (type (;0;) (func (param i64) (result i32 i64)))
```

This transformation generates random function signatures with a random number of parameters and results count. This rewriting rule does not affect the runtime behavior of the variant. It also guarantees that the index of the already defined types is consistent after the addition of a new type. This is because Wasm programs cannot access or use a type definition during runtime, they are only used to validate the signature of a function during compilation and validation from the host engine. From the security perspective, this transformation prevents against static binary analysis. For example, to avoid malware detection based on signature set [8].

Add function: The function and code sections of a Wasm binary contain function declarations and the code body of the declared functions, respectively. WASM-MUTATE add new functions, through mutations in the two mentioned sections. To add a new function, WASM-MUTATE creates a random type signature. Then, the random

function body is created. The body of the function consists of returning the default value of the result type. The following example illustrates this rewriting rule.

```
LHS (module
   (type (;0;) (func (param i32 f32) (result i64)))
```

```
RHS (module
   (type (;0;) (func (param i32 f32) (result i64)))
+_______(func (;0;) (type 0) (param i32 f32) (result i64)
+__________i64.const 0)
```

WASM-MUTATE never adds a call instruction to this function. So in practice, the new function is never executed. Therefore, executing both, the original binary and the mutated one with the same input, lead to the same final state. This strategy follows the work of Cohen, advocating the insertion of harmless 'garbage' code into a program. These transformations do not impact the program's functionality; they increase its static complexity.

Remove dead code: WASM-MUTATE can randomly remove dead code. In particular WASM-MUTATE removes: *functions, types, custom sections, imports, tables, memories, globals, data segments and elements* that can be validated as dead code with guarantees. For instance, to delete a memory declaration, the binary code must not contain a memory access operation. Separate mutators are included within WASM-MUTATE for each of the aforementioned elements. For a more concrete example, the following listing illustrates the case of a function removal.

```
LHS (module (type (func)))
```

```
RHS - (module (import "" "" (func)))

Cond The removed function is not called, it is not exported, and it is
       not in the binary _table.
```

When removing a function, WASM-MUTATE ensures that the resulting binary remains valid and semantically identical to the original binary: it checks that the deleted function was neither called within the binary code nor exported in the binary external interface. As exemplified above, WASM-MUTATE might also eliminate a function import while removing the function.

Eliminating dead code serves a dual purpose: it minimizes the attack surface available to potential malicious actors [1] and strengthens the resilience of security protocols. For instance, it can obstruct signature-based identification [8]. With Narayan and colleagues having demonstrated the feasibility of Return-Oriented Programming (ROP) attacks [36], the removal of dead code is able to stop jumps to harmful behaviors within the binary. On the other hand, the act of removing dead code reduces the binary's size, improving its non-functional properties, in particular bandwidth constraints.

Edit custom sections: The custom section in WebAssembly is used to store metadata, such as the name of the compiler that produces the binary or the symbol information for debugging. Thus, this section does not affect the execution of the Wasm program. WASM-MUTATE includes one mutator to edit custom sections. This is exemplified in the following rewriting rule.

```
LHS (module
...
   - (@custom "CS42" "zzz...")
```

```
RHS (module
...
   + (@custom "CS42" "xxx...")
```

The *Edit Custom Section* transformation operates by randomly modifying either the content or the name of the custom section. As illustrated by Cabrera-Arteaga et al. [8], such a rewriting strategy also acts as a potent deterrent against compiler identification techniques. Furthermore, it can also be employed in an innovative manner to emulate the characteristics of a different compiler, *masquerading* as another compilation source. This strategy ultimately aids in shrinking the identification and fingerprinting surface accessible to potential adversaries, hence enhancing overall system security, or to make it a moving target.

If swapping: In WebAssembly, an if-construction consists of a consequence and an alternative. The branching condition is executed right before the `if` instruction; if the value at the top of the stack is greater than `0`, then the consequence-code is executed, otherwise the alternative-code is run. The *if swapping* rewriting swaps the consequence and alternative codes of an if-construction.

To swap an if-construction in WebAssembly, WASM-MUTATE inserts a negation of the value at the top of the stack right before the `if` instruction. In the following rewriting rule we show how WASM-MUTATE performs this rewriting.

```
LHS (module
   (func ...) (
condition C
      (if A else B end)
   )
)
```

```
RHS (module
   (func ...) (
condition C
i32.eqz
      (if B else A end)
   )
)
```

The consequence and alternative codes are annotated with the letters `A` and `B`, respectively. The condition of the if-construction is denoted as `C`. The negation of the condition is achieved by adding the `i32.eqz` instruction in the RHS part of the rewriting rule. The `i32.eqz` instruction compares the top value of the stack with zero, pushing the value `1` if the comparison is true. Some if-constructions may not have either a consequence or an alternative code. In such cases, WASM-MUTATE replaces the missing code block with a single `nop`

instruction. In the context of ROP [36], this transformation can protect a victim binary to be exploited.

Loop Unrolling: Loop unrolling is a technique employed to enhance the performance of programs by reducing loop control overhead [14]. WASM-MUTATE incorporates a loop unrolling transformation and utilizes the Abstract Syntax Tree (AST) of the original Wasm binary to identify loop constructions.

When WASM-MUTATE selects a loop for unrolling, its instructions are divided by first-order breaks, which are jumps to the loop's start. This separation ensures that branching instructions controlling the loop body do not require label index adjustments during unrolling. The same holds true for instructions continuing to the next loop iteration. As the loop unrolling process unfolds, a new Wasm block is created to encompass both the duplicated loop body and the original loop. Within this newly established block, the previously separated groups of instructions are copied. These replicated groups of instructions mirror the original ones, except for branching instructions jumping outside the loop body, which need their jumping indices increased by one. This modification is required due to the introduction of a new `block ... end` scope around the loop body, which affects the scope levels of the branching instructions.

In the following text we illustrate the rewriting rule for a function that contains a loop.

```
LHS (module
    (func ...) (
        (loop   A br_if 0   B end)
    )
)
```

```
RHS (module
    (func ...) (
        (block
            (block A'   br_if 0   B'   br 1   end)
            (loop   A'   br_if 0   B'   end)
        end)
    )
)
```

The loop in the LHS part features a single first-order break, indicating that its execution will cause the program to continue iterating through the loop. The loop body concludes right before the `end` instruction, which highlights the point at which the original loop breaks and resumes program execution. Upon selecting the loop for unrolling, its instructions are divided into two groups, labeled `A` and `B`. As illustrated in the RHS part, the unrolling process entails creating two new Wasm blocks. The outer block encompasses both the original loop structure and the duplicated loop body, while the inner blocks, denoted as `A'` and `B'`, represent modifications of the jump instructions in groups `A` and `B`, respectively. Notice that, any jump instructions within `A'` and `B'` that originally leaped outside the loop must have their jump indices incremented by one. This adjustment accounts for the new block scope introduced around the loop body during the unrolling process. Furthermore, an unconditional branch is placed at the end of the unrolled loop iteration's body. This ensures that if the loop body does not continue, the tool breaks out of the scope instead of proceeding to the non-unrolled loop.

Loop unrolling enhances resistance to static analysis while maintaining the original performance [38]. In particular, Crane et al. [13] have validated the effectiveness of adding and modifying jump instructions against Function-Reuse attacks. Our rewriting rule has the same advantages, it unrolls loops while 1) incorporating new jumps and 2) editing existing jumps, as it can be observed with the addition of the `br_if`, `end`, and `br` instructions.

Peephole: This transformation category is about rewriting instruction sequences within function bodies, signifying the most granular level of rewriting. We implement 125 rewriting rules for this group in WASM-MUTATE. We include rewriting rules that affects the memory of the binary. For example, we include rewriting rules that creates random assignments to newly created global variables. For these rules, we incorporate several conditions, denoted by `Cond`, to ensure successful replacement. These conditions can be utilized interchangeably and combined to constrain transformations (see subsection 3.3).

For instance, WASM-MUTATE is designed to guarantee that instructions marked for replacement are deterministic. We specifically exclude instructions that could potentially cause undefined behavior, such as function calls, from being mutated. For this rewriting type, WASM-MUTATE only alters stack and memory operations, leaving the control frame labels unaffected.

The peephole category rewriting rules are meticulously designed and manually verified. An instance of such streamlined transformation can is illustrated in subsection 2.3, `( x i32.or x, x, {})` implies that the `LHS` 'x' is to be replaced by an idempotent bitwise `i32.or` operation with itself, in the absence of any specific conditions. Therefore, this category continues to uphold the benefits previously discussed under the *Remove Dead Code* category.

3.3. E-graphs for WebAssembly

We build WASM-MUTATE on top of e-graphs. An e-graph is a graph data structure utilized for representing rewriting rules [9] and their chaining. In an e-graph, there are two types of nodes: e-nodes and e-classes. An e-node represents either an operator or an operand involved in the rewriting rule, while an e-class denotes the equivalence classes among e-nodes by grouping them, i.e., an e-class is a virtual node compound of a collection of e-nodes. Thus, e-classes contain at least one e-node. Edges within the graph establish operator-operand equivalence relations between e-nodes and e-classes.

In WASM-MUTATE, the e-graph is automatically built from a WebAssembly program by analyzing its expressions and operations through its data flow graph. Then, each unique expression, operator, and operand are transformed into e-nodes. Based on the input rewriting rules, the equivalent expressions are detected, grouping equivalent e-nodes into e-classes. During the detection of equivalent expres-

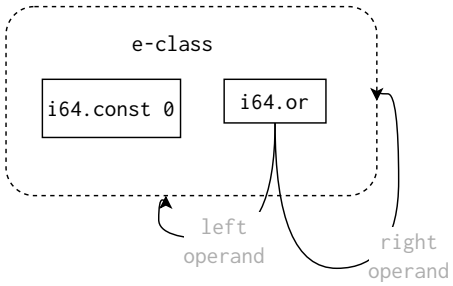

Figure 2: e-graph for idempotent bitwise-or rewriting rule. Solid lines represent operand-operator relations, and dashed lines represent equivalent class inclusion.

sions, new operators could be added to the graph as e-nodes. Finally, e-nodes within an e-class are connected with edges to represent their equivalence relationships.

For example, let us consider one program with a single instruction that returns an integer constant, `i64.const 0`. Let us also assume a single rewriting rule, `(x, x i64.or x, x instanceof i64)`. In this example, the program's control flow graph contains just one node, representing the unique instruction. The rewriting rule represents the equivalence for performing an `or` operation with two equal operands. Figure 2 displays the final e-graph data structure constructed out of this single program and rewriting rule. We start by adding the unique program instruction `i64.const 0` as an en e-node (depicted by the leftmost solid rectangle node in the figure). Next, we generate e-nodes from the rewriting rule (the rightmost solid rectangle) by introducing a new e-node, `i64.or`, and creating edges to the `x` e-node. Following this, we establish equivalence. The rewriting rule combines the two e-nodes into a single e-class (indicated by the dashed rectangle node in the figure). As a result, we update the edges to point to the `x` symbol e-class.

Willsey et al. illustrate that the extraction of code fragments from e-graphs can achieve a high level of flexibility, especially when the extraction process is recursively defined through a cost function applied to e-nodes and their operands. This approach guarantees the semantic equivalence of the extracted code [48]. For example, to obtain the smallest code from an e-graph, one could initiate the extraction process at an e-node and then choose the AST with the smallest size from among the operands of its associated e-class [35]. When the cost function is omitted from the extraction methodology, the following property emerges: *Any path traversed through the e-graph will result in a semantically equivalent code variant*. This concept is illustrated in Figure 2, where it is possible to construct an infinite sequence of "or" operations. In the current study, we leverage this inherent flexibility to generate mutated variants of an original program. The e-graph offers the option for random traversal, allowing for the random selection of an e-node within each e-class visited, thereby yielding an equivalent expression.

We propose and implement the following algorithm to randomly traverse an e-graph and generate semantically

Algorithm 1 e-graph traversal algorithm.

```
1:  procedure TRAVERSE(egraph, eclass, depth)
2:      if depth = 0 then
3:          return smallest_tree_from(egraph, eclass)
4:      else
5:          nodes ← egraph[eclass]
6:          node ← random_choice(nodes)
7:          expr ← (node, operands = [])
8:          for each child ∈ node.children do
9:              subexpr ← TRAVERSE(egraph, child, depth − 1)
10:             expr.operands ← expr.operands ∪ {subexpr}
11:         return expr
```

equivalent program variants, see 1. It receives an e-graph, an e-class node (initially the root's e-class), and the maximum depth of expression to extract. The depth parameter ensures that the algorithm is not stuck in an infinite recursion. We select a random e-node from the e-class (lines 5 and 6), and the process recursively continues with the children of the selected e-node (line 8) with a decreasing depth. As soon as the depth becomes zero, the algorithm returns the smallest expression out of the current e-class (line 3). The subexpressions are composed together (line 10) for each child, and then the entire expression is returned (line 11). To the best of our knowledge, WASM-MUTATE, is the first practical implementation of random e-graph traversal for WebAssembly.

Let's demonstrate how the proposed traversal algorithm can generate program variants with an example. We will illustrate Algorithm 1 using a maximum depth of 1. Listing 5 presents a hypothetical original Wasm binary to mutate. In this example, the developer has established two rewriting rules: `(x, x i32.or x, x instanceof i32)` and `(x, x i32.add 0, x instanceof i32)`. The first rewriting rule represents the equivalence of performing an `or` operation with two equal operands, while the second rule signifies the equivalence of adding 0 to any numeric value. By employing the code and the rewriting rules, we can construct the e-graph depicted in Figure 3. The figure demonstrates the operator-operand relationship using arrows between the corresponding nodes.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        i64.const 1)
)
```

Listing 5: Wasm function.

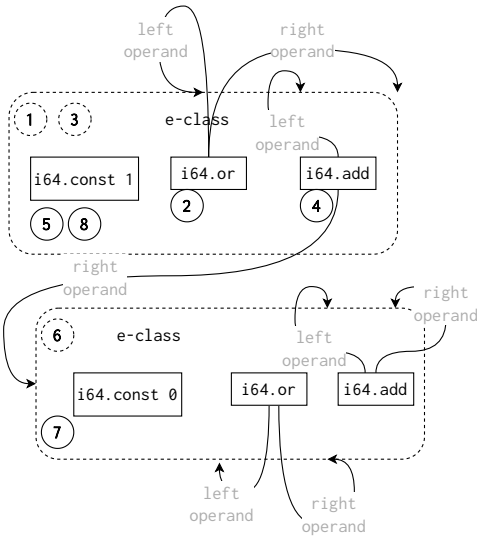

Figure 3: e-graph built starting in the first instruction of Listing 5.

```
(module
    (type (;0;) (func (param i32 f32) (result i64)))
    (func (;0;) (type 0) (param i32 f32) (result i64)
        (i64.or (
            (i64.add (
                i64.const 0
                i64.const 1
            ))
            i64.const 1
        ))
    )
```

Listing 6: Random peephole mutation using egraph traversal for Listing 5 over e-graph Figure 3. The textual format is folded for better understanding.

In Figure 3, we annotate the various steps of Algorithm 1 for the scenario described above. Algorithm 1 begins at the e-class containing the single instruction `i64.const 1` from Listing 5. It then selects an equivalent node in the e-class (2), in this case, the `i64.or` node, resulting in: `expr = i64.or 1 r`. The traversal proceeds with the left operand of the selected node (3), choosing the `i64.add` node within the e-class: `expr = i64.or (i64.add 1 r) r`. The left operand of the `i64.add` node is the original node (5): `expr = i64.or (i64.add i64.const 1 r) r`. The right operand of the `i64.add` node belongs to another e-class, where the node `i64.const 0` is selected (6)(7): `expr = i64.or (i64.add i64.const 1 i64.const 0) r`. In the final step (8), the right operand of the `i64.or` is selected, corresponding to the initial instruction e-node, returning: `expr = i64.or (i64.add i64.const 1 i64.const 0)i64.const 1` The traversal result applied to the original Wasm code can observed in Listing 6.

3.4. WASM-MUTATE in practice

In practice, WASM-MUTATE serves as a module within a broader process. This process starts from a WebAssembly binary as input and iterates over the variants generated by WASM-MUTATE in order to provide guarantees. In particular, it ensures that the output variant exhibits a different machine code per the JIT engine that executes it and unique execution traces when running. This process is explicitly laid out in Algorithm 2. One of the key elements in this algorithm is line 8, which activates WASM-MUTATE's diversification engine.

The algorithm starts by running the original WebAssembly program and recording its original execution traces, as denoted in line 5. These initial traces act as a reference for evaluating subsequent variants. An budget-based loop then initiates, as marked by lines 8 and 9, aiming to apply a series of code transformations. Upon the successful creation of a unique variant, line 11 triggers a JIT compilation within the WebAssembly engine. This step compiles the variant into machine code. The algorithm next assesses whether this machine code diverges from the original, thus confirming the actual diversity. If this condition is satisfied, the algorithm executes the variant to collect its low-level execution traces. The loop ends when a variant is found with new traces that are distinct from the original, as validated in line 15. The algorithm then returns the generated variant, wich guarantees that both diversified machine code and traces are different from the original.

Algorithm 2 WASM-MUTATE in practice.

```
1:  procedure DIVERSIFY(originalWasm, engine)
2:  Input:      ▷ A WebAssembly binary to diversify and a WebAssembly engine.
3:  Output:     ▷ A statically unique and behaviourally different WebAssembly variant.
4:
5:      originalTrace ← engine.execute(originalWasm)
6:      wasm ← originalWasm
7:      while true do
8:          variantWasm ← WASM-MUTATE(wasm)
9:          wasm ← variantWasm // we stack the transformation
10:         if variantWasm is unique then
11:             variantJIT ← engine.compile(variantWasm)
12:             if variantJIT is unique then
13:
14:                 trace ← engine.execute(variantJIT)
15:                 if trace ≠ originalTrace then
16:                     return variantWasm
```

3.5. Implementation

WASM-MUTATE is implemented in Rust, comprising approximately, 10 thousands lines of Rust code. We leverage the capabilities of the wasm-tools project of the bytecodealliance for parsing and transforming WebAssembly binary code. Specifically, we utilize the wasmparser and wasm-encoder modules for parsing and encoding Wasm binaries, respectively. The implementation of WASM-MUTATE is publicly availabel for future research and can be found at `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-mutate`.

Source	Program	RQ	#F	# Ins.	Attack
CROW [7]	303	RQ1, RQ2	7-103	170-36023	N/A
Swivel [36]	btb_breakout	RQ3	16	743	Spectre branch target buffer (btb)
Swivel [36]	btb_leakage	RQ3	16	297	Spectre branch target buffer(btb)
Safeside [36, 20]	ret2spec	RQ3	2977	378894	Spectre Return Stack Buffer (rsb)
Safeside [36, 20]	pht	RQ3	2978	379058	Spectre Pattern History Table (pht)

Table 1
WebAssembly dataset used to evaluate WASM-MUTATE. Each row in the table corresponds to programs, with the columns providing: where the program is sourced from, the number of programs, research question addressed, function count, the total number of instructions found in the original WebAssembly program and the type of attack that the original program was subjected to.

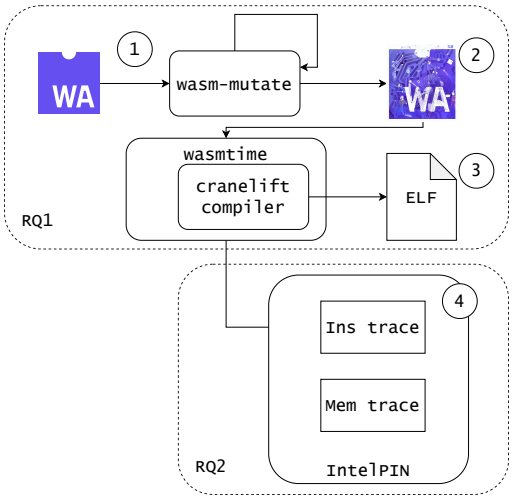

Figure 4: Protocol to answer RQ1 and RQ2

4. Evaluation

In this section, we outline our methodology for evaluating WASM-MUTATE. Initially, we introduce our research questions and the corpus of programs that we utilize for the assessment of WASM-MUTATE. Next, we elaborate on the methodology for each research question. For the sake of reproducibility, our data and experimenting pipeline are publicly available at `https://github.com/ASSERT-KTH/tawasco`. Our experiments are conducted in Standard F4s-v2(Skylake) Azure machines with 4 virtual cpus and 8GiB memory per instance.

RQ1: **To what extent are the program variants generated by** WASM-MUTATE **statically different from the original programs?** We check whether the WebAssembly binary variants radpidly produced by WASM-MUTATE are different from the original WebAssembly binary. Then, we assess whether the x86 machine code produced by wasmtime engine is also different.

RQ2: **How fast can** WASM-MUTATE **generate program variants that exhibit different execution traces?** To assess the versatility of WASM-MUTATE, we also examine the presence of different behaviors in the generated variants. Specifically, we measure the speed at which WASM-MUTATE generates variants with distinct machine code instruction traces and memory access patterns.

RQ3: **To what extent does** WASM-MUTATE **prevent side-channel attacks on WebAssembly programs?** Diversification being an option to prevent security issues, we assess the impact of WASM-MUTATE in preventing one class of attacks: cache attacks (Spectre).

4.1. Corpora

We answer our research questions with a corpus of 307 programs (303 + 4). These programs are summarized in Table 1. Each row in the table corresponds to the used programs, with the columns providing: where the program is sourced from, the number of programs, research question addressed, function count, the total number of instructions found in the original WebAssembly program and the type of attack that the original program was subjected to.

We answer RQ1 and RQ2 with corpus of programs from Cabrera et.al. [7], it is shown in the first row of Table 1. The corpus contains 303. The corpus contains programs for a range of tasks, from simple ones, such as sorting, to complex algorithms like a compiler lexer. The number of functions for each program ranges from 7 to 103 and, the number of total instructions ranges from 170 to 36023. All programs in corpus: 1) do not require input from user, i.e., do not functions like `scanf`, 2) terminate, 3) are deterministic, i.e., given the same input, provide the same output and 4) compile to WebAssembly using `wasi-clang` to compile them.

We answer RQ3 with four WebAssembly programs and three Spectre attack scenarios, from the Swivel project [36]. These programs are summarized in the final four rows of our corpus table. The first two programs are manually crafted and contain 16 functions, with instruction counts of 743 and 297, respectively. These binaries are specifically designed to perform the Spectre branch target attack. The third and fourth programs, documented in rows four and five, comefrom the Safeside project [20]. Unlike the first two, these binaries are significantly larger, each containing nearly 3000 functions and more than 300000 instructions. They are utilized for conducting the Spectre Return Stack (RSB) and Spectre Pattern History (PHT) attacks [28].

There is a notable difference in the number of functions and instructions between the first pair of Swivel binaries and the latter pair. This disparity can be attributed to the varying compilation processes applied to these WebAssembly binaries. The three attack scenarios are described in details in subsection 4.4.

4.2. Protocol for RQ1

With RQ1, we assess the ability of WASM-MUTATE to generate WebAssembly binaries that are different from the original program, including after their compilation to x86 machine code. In Figure 4 we show the steps we follow to answer RQ1. We run WASM-MUTATE on our corpus of 303 original C programs (step ① in figure). To generate the variants: 1) we start with one original and pass it to WASM-MUTATE to generate a variant; 2) the variant and the original program form a population of programs; 3) we randomly select a program from this population and pass it to WASM-MUTATE to generate a variant, which we add to the population; 4) we then restart the process in the previous step. to stack more mutations This procedure is carried out for a duration of 1 hour. The final outcome (step ② in figure) is a population with a number of stacked transformations, all starting from an original WebAssembly program. We then count the number of unique variants in the population. We compute the sha256 hash of each variant bytestream in order and define the population size metric as:

Metric 1. *Population_size(P): Given an original WebAssembly program P, a generated corpus of WebAssembly programs* $V = \{v_1, v_2, ..., v_N\}$ *where* v_i *is a variant of P, the population size is defined as:*

$$|set(\{sha256(v_1), ...sha256(v_N)\})| \; \forall v_i \in V$$

Since WebAssembly binaries may be further transformed into machine code before they execute, we also check that this additional transformations preserve the difference introduces by WASM-MUTATE in the WebAssembly binary. We use the wasmtime JIT compiler, cranelift, with all available optimizations, to generate the x86 binaries for each WebAssembly program and its variants (step ③ in figure). Then, we calculate the number of unique variants machine code representation for wasmtime. Counting the number of unique machine code, we compute the diversification preservation ratio:

Metric 2. *Ratio of preserved variants: Given an original WebAssembly program P and its population size as defined in Metric 1 and the JIT compiler C, we defined the ratio of preserved variants as:*

$$\frac{|set(\{sha256(C(v_1)), ...sha256(C(v_N))\})|}{Population\_size\ (P)} \; \forall v_i \in V$$

If $sha256(P_1) \neq sha256(P_2)$ and $sha256(C(P_1))) \neq sha256(C(P_2))$, this means that both programs are still different after being compiled to machine code, and this means that the cranelift compiler has not removed the transformations made by WASM-MUTATE.

Note that the protocol described earlier can be mapped to Algorithm 2. For instance, to measure population size for each tested program, one could measure how often the execution of Algorithm 2 reaches line 11. Similarly, to assess the level of preservation, one could track the frequency with which the algorithm arrives at line 13.

4.3. Protocol for RQ2

For RQ2, we evaluate how fast WASM-MUTATE can generate variants that offer distinct traces compared with the original program. We start by collecting the traces of the original program when executed in wasmtime. While continuously generating variants with random stacked transformations, we collect the execution traces of the variants as well. We record the time passed until we generate a variant that offers different execution traces, according to two types of traces: machine code instructions and memory accesses. This process can be seen in the enclosed square of Figure 4, annotated with RQ2.

We gather the instructions and memory traces utilizing IntelPIN [33, 16] (step ④ in the figure). To only collect the traces of the WebAssembly execution with a wasmtime engine, we pause and resume the collection as the execution leaves and re-enters the WebAssembly code, respectively. We implement this filtering with the built-in hooks of wasmtime. In addition, we disable ASLR on the machine where the variants are executed. This latter action ensures that the placement of the instructions in memory is deterministic. Examples of the traces we collect can be seen in Listing 7 and Listing 8 for memory and instruction traces, respectively.

```
[Writ] 0x555555ed1570 size=4 value=0x10dd0
[Read] 0x555555ed1570 size=4 value=0x10dd0
```

Listing 7: Memory trace with two events out of IntelPIN for the execution of a WebAssembly program with wasmtime. Trace events record: the type of the operation, read or write, the memory address, the number of bytes affected and the value read or written.

```
[I] mov rdx, qword ptr [r14+0x100]
[I] mov dword ptr [rdx+0xe64], ecx
```

Listing 8: Instructions trace with two events out of IntelPIN for the execution of a WebAssembly program with wasmtime. Each event records the corresponding machine code that executes.

In the text below, we outline the metric used to assess how fast WASM-MUTATE can generate variants that provide different execution traces.

Metric 3. *Time until different trace: Given an original WebAssembly program P, and an its execution trace* T_1*, the time until different trace is defined as the time between the diversification process starts and the when the variant V is generated with execution trace* T_2*, and* $T_1 \neq T_2$*.*

Notice that the previously defined metric is instantiated twice, for instructions and memory type of events.

Refering to Algorithm 2, we quantify the elapsed time between line 6 and line 16 to obtain the time it takes for WASM-MUTATE to generate a unique WebAssembly variant producing different execution traces.

4.4. Protocol for RQ3

To answer RQ3, we apply WASM-MUTATE to the same security WebAssembly programs used by Narayan et al. to evaluated Swivel's ability at protecting WebAssembly programs against side-channel attacks [36]. The four cache timing side-channel attacks are presented in detail in subsection 4.1. The specific binary and its corresponding attack can be appreciated in Table 1. We evaluate to what extent WASM-MUTATE can prevent such attacks. In the following text, we describe the attacks we replicate and evaluate in order of answering RQ3.

Narayan and colleagues successfully bypass the control flow integrity safeguards, using speculative code execution as detailed in [28]. Thus, we use the same three Spectre attacks from Swivel: 1) The Spectre Branch Target Buffer (btb) attack exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target. 2) The Spectre Pattern History Table (pht) takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition. 3) The Spectre Return Stack Buffer (ret2spec) attack exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions. Each attack methodology relies on the extraction of memory bytes from another hosted WebAssembly binary that executes in parallel.

For each of the four WebAssembly binaries introduced in subsection 4.1, we generated a maximum of 1000 random stacked transformations utilizing 100 distinct seeds. This resulted in a total of 100,000 variants for each original WebAssembly binary. We then assess the success rate of attacks across these variants by measuring the bandwidth of the exfiltrated data, that is: the rate of correctly leaked bytes per unit of time. We then count the correctly exfiltrated bytes and divided them by the variant program's execution time.

Notice that, the bandwidth metric captures not only whether the attacks are successful or not, but also the degree to which the data exfiltration is hindered. For instance, a variant that continues to exfiltrate secret data but does so over an impractical duration would be deemed as having been hardened. For this, we state the bandwidth metric in the following definition :

Metric 4. *Bandwidth: Given data* $D = \{b_0, b1, ..., b_C\}$ *being exfiltrated in time* T *and* $K = k_1, k_2, ..., k_N$ *the collection of correct data bytes, the bandwidth metric is defined as:*

$$\frac{|b_i \text{ such that } b_i \in K|}{T}$$

5. Experimental Results

5.1. To what extent are the program variants generated by WASM-MUTATE statically different from the original programs?

To address RQ1, we utilize WASM-MUTATE to process the original 303 programs from [7]. WASM-MUTATE is set to generate variants with a timeout of one hour for each individual program. Following this, we assess the sizes of their variant populations as well as their corresponding preservation ratio (Refer to Metric 1 and Metric 2 for more details).

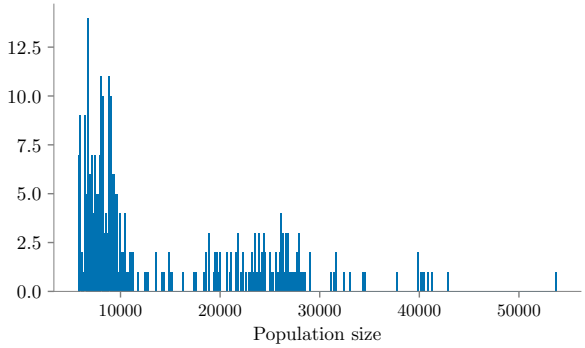

Figure 5: RQ1: Number of unique WebAssembly programs generated by WASM-MUTATE in 1 hour out of each program of the corpus.

In Figure 5, we show the distribution of the population size generated out of WASM-MUTATE. WASM-MUTATE successfully diversifies all 303 original programs, yielding a diversification rate of 100%. Within an hour, WASM-MUTATE demonstrates its impressive efficiency and effectiveness by producing a median of 9500 unique variants for the 303 original programs. The largest population size observed is 53816, while the smallest is 5716. There are several factors contributing to large population sizes.

WASM-MUTATE can diversify functions within WASI-libc. Despite the relatively low function count in the original source code, WASM-MUTATE creates thousands of distinct variants in the function of the incorporated libraries. This feature improves over methods that can only diversify the original source code processed through the LLVM compilation pipeline [7].

We have observed a significant variation in the population size out of WASM-MUTATE between different programs, ranging by several thousand variants (from a maximum of 53816 variants to a minimum of 5716 variants). This disparity is attributed to: the non-deterministic nature of WASM-MUTATE and 2) the characteristics of the program. WASM-MUTATE mutates a randomly selected portion of a program. If the selected instruction is determined to be non-deterministic, despite the transformation being semantically equivalent, WASM-MUTATE discards the variant and moves on to another random transformation. For instance, if the instruction targeted for mutation is a function call, WASM-MUTATE proceeds to the next one. This process, in conjunction with the unique characteristics of each program, results in a varying population size. For example, an input binary with a high number of function calls would lead to a greater number of trials and errors, slowing down the generation of variants, thereby resulting in a smaller overall population size for 1 hour of WASM-MUTATE execution.

As stated in subsection 4.2, we also assess static diver-

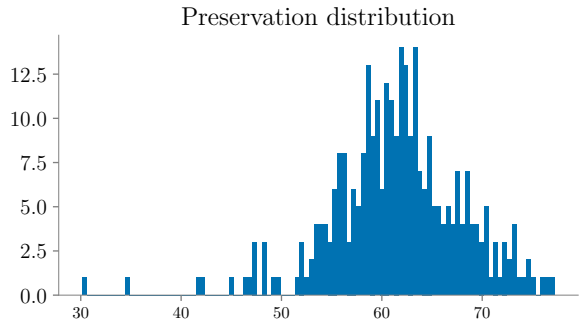

Figure 6: RQ1: Distribution of the ratio of wasmtime preserved variants.

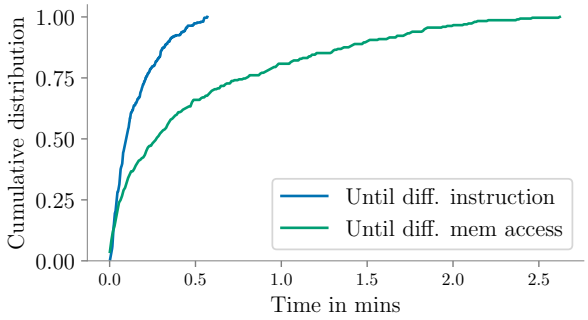

Figure 7: RQ2: Cumulative distribution for time until different trace. In blue for different machine code instructions, in green for different memory traces. The X-axis marks time in minutes, and the Y-axis shows the ratio of programs from 303 for which WASM-MUTATE created a variant within that time.

sification with Metric 2 by calculating the preservation ratio of variant populations. Figure 6 presents the distribution of preservation ratios for the cranelift compiler of wasmtime. We have observed a median preservation ratio of 62%. On the one hand, we have observed that there is no correlation between population size and preservation ratio. In other words, having a larger population size does not necessarily lead to a higher preservation ratio. On the other hand, the phenomena of non-preserved variants can be explained as follows. Factors such as custom sections are often disregarded by compilers. Similarly, bloated code plays a role in this context. For instance, WASM-MUTATE generates certain variants with unused types or functions, which are then detected and eliminated by cranelift. Yet, note that even when working with the smallest population size and the lowest preservation percentage, the number of unique machine codes can still encompass thousands of variants.

Answer to RQ1: WASM-MUTATE generates WebAssembly variants for the 303 programs, which are different from the original program. Within a one-hour diversification budget, WASM-MUTATE synthesizes more than 9000 unique variants per program on average. 62% of the variants remain different after machine-code compilation. WASM-MUTATE is good at producing a large number of WebAssembly program variants.

5.2. How fast can WASM-MUTATE generate program variants that exhibit different execution traces?

To answer question RQ2, we measure how long it takes to generate one variant that exhibits execution traces that are different from the original. In Figure 7, we display a cumulative distribution plot showing the time required for WASM-MUTATE to generate variants with different traces, in blue for machine code instructions and green for memory traces. The X-axis marks time in minutes, and the Y-axis shows the ratio of programs from 303 for which WASM-MUTATE created a variant within that time. For all original program, WASM-MUTATE succeeds in generating one variant with different traces comparing to the original program, either in machine code instructions or memory access, ie both cumlative distributions readch 100The shortest time to generate a variant with different machine code instruction traces is 0.12 seconds, and for different memory traces, it is 0.06 seconds. In the slowest scenarios, WASM-MUTATE takes under 1 minute for different machine code instruction traces and less than 3 minutes for different memory traces. Overall, WASM-MUTATE takes a median of 5.4 seconds and 12.6 seconds in generating variants with different machine code instructions and different memory instructions respectively.

The use an e-graph random traversal is the key factor for such a fast generation process. Once WASM-MUTATE locates a modifiable instruction within the binary and constructs its corresponding e-graph, traversal is virtually instantaneous. However, the time efficiency of variant generation is not consistent across all programs, as illustrated in Figure 7. This variation primarily stems from the varying complexities of the programs under analysis, as previously mentioned in subsection 5.1. Interestingly, WASM-MUTATE may attempt to build e-graphs from instructions that, while not inherently leading to undefined behavior, are part of a data flow graph that could. For example, the data flow graph might be dependent on a function call. Although transforming undefined behavioural instructions is deactivated by default in WASM-MUTATE to maintain functional equivalence with the original code, the process of attempting to construct such e-graphs can extend the duration of the diversification pass. As a result, WASM-MUTATE may require multiple attempts to successfully create and traverse an e-graph, impacting the rate at which it generates behaviorally distinct variants. This phenomenon is particularly noticeable in original programs that have a high frequency of function calls.

In average, WASM-MUTATE takes three times longer to synthesize unique memory traces than it does to generate different instruction traces (as it can be observed in how the green plot of the figure is skewed to the right). The main reason for this difference is the limited set of rewriting rules that specifically focus on memory operations. WASM-

MUTATE includes more rules for manipulating code, which increases the odds of generating a variant with diverse machine code instructions. Additionally, the variant creation process halts and restarts with alternative rewriting rules if WASM-MUTATE detects that the selected code for transformation could result in unpredictable behavior.

We have identified four primary factors explaining why execution traces differs overall. First, alterations to the binary layout inherently impact both machine code instruction traces and memory accesses within the program's stack. In particular, WASM-MUTATE creates variants that change the return addresses of functions, leading to divergent execution traces, including those related to memory access. Second, our rewriting rules incorporate artificial global values into WebAssembly binaries. Since these global variables are inherently manipulated via the stack, their access inevitably generate divergent memory traces. Third, WASM-MUTATE injects 'phantom' instructions which do not aim to modify the outcome of a transformed function during execution. These intermediate calculations trigger the spill/reload component of the runtime, varying spill and reload operations. In the context of limited physical resources, these operations temporarily store values in memory for later retrieval and use, thus creating unique memory traces. Finally, certain rewriting rules implemented by WASM-MUTATE replicate fragments of code, e.g., performing commutative operations. These code segments may contain memory accesses, and while neither the memory addresses nor their values change, the frequency of these operations does. Overall, these findings influence the diversity of execution traces among the generated variants.

Answer to RQ2: WASM-MUTATE generates variants with distinct machine code instructions and memory traces for all tested programs. The quickest time for generating a variant with a unique machine code trace is 0.12 seconds, and for divergent memory traces, it's best time is 0.06 seconds. On average, the median time required to produce a variant with distcint traces stands at 5.4 seconds for unique machine code traces and 16.2 seconds for different memory traces. These metrics indicate that WASM-MUTATE is suitable for fast-moving target defense strategies, capable of generating a new variant in well under a minute [6]. To the best of our knowledge, WASM-MUTATE is the fastest diversification engine for WebAssembly.

5.3. To what extent does WASM-MUTATE prevent side-channel attacks on WebAssembly programs?

To answer RQ3, we execute WASM-MUTATE on four distinct binaries WebAssembly susceptible to Spectre related attacks. Each of the four programs is transformed with one of for 100 different seeds and up to 1000 stacked transformations. We assess the resulting impact of the attacks as outlined in 4.4. The analysis encompasses a total of 4×100×1000 binaries, which also includes the original four.

Figure 8 offers a graphical representation of WASM-MUTATE's influence on the Swivel original programs and their attacks. Each plot corresponds to one original WebAssembly binary and the attack it undergoes: btb_breakout, btb_leakage, ret2spec, and pht. The Y-axis represents the exfiltration bandwidth (see Metric 4). The bandwidth of the original binary under attack is marked as a blue dashed horizontal line. In each plot, the variants are grouped in clusters of 100 stacked transformations. These are indicated by green dots and lines. The dot signifies the median bandwidth for the cluster, while the line represents the interquartile range of the group's bandwidth.

For btb_breakout and btb_leakage, WASM-MUTATE demonstrates effectiveness, generating variants that leak less information than the original in 78% and 70% of the cases, respectively. For these particular binaries, a significant reduction in exfiltration bandwidth to zero is noted after 200 stacked transformations. This means that with a minimum of 200 stacked transformations, WASM-MUTATE can create variants that are completely resistant to the original attack. For the ret2spec and pht scenarios, the produced variants consistently exhibit lower bandwidth than the original in 76% and 71% of instances, respectively. As depicted in the plots, the exfiltration bandwidth diminishes following the application of at least 100 stacked transformations.

This success is explained by the fact that WASM-MUTATE synthesizes variants that effectively alter memory access patterns. Specifically, it does so by amplifying spill/reload operations, injecting artificial global variables, and changing the frequency of pre-existing memory accesses. These transformations influence the WebAssembly program's memory, causing disruption to cache predictors. As a result, these alterations contribute to a reduction in exfiltration bandwidth.

Furthermore, many attacks rely on a timer component to measure cache access time for memory, and disrupting this component effectively impairs the attack's effectiveness. This strategy of dynamic alteration has also been employed in other scenarios. For instance, to counter potential timing attacks, Firefox randomizes its built-in JavaScript timer [42]. WASM-MUTATE applies the same strategy by interspersing instructions within the timing steps of WebAssembly variants. In Listing 9 and Listing 10, we demonstrate WASM-MUTATE's impact on time measurements. The former illustrates the original time measurement, while the latter presents a variant with WASM-MUTATE-inserted operations amid the timing.

```
;; Code from original btb_breakout
...
(call $readTimer)
(set_local $end_time)
... access to mem
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 9: Wasm timer used in btb_breakout program.

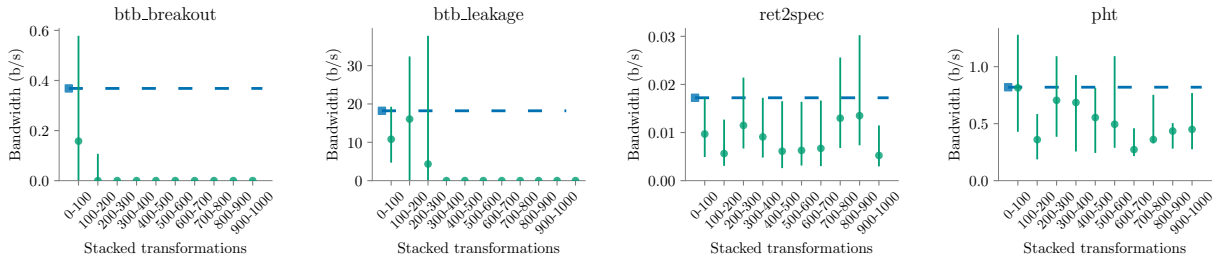

Figure 8: Visual representation of WASM-MUTATE's impact on Swivel's original programs. The Y-axis denotes exfiltration bandwidth, with the original binary's bandwidth under attack highlighted by a blue marker and dashed line. Variants are clustered in groups of 100 stacked transformations, denoted by green dots (median bandwidth) and lines (interquartile bandwidth range). Overall, for all 100000 variants generated out of each original program, 70% have less data leakage bandwidth.

```
;; Variant code
...
(call $readTimer)
(set_local $end_time)
<inserted instructions>
... access to mem
<inserted instructions>
(i64.sub (get_local $end_time ) (get_local $start_time))
(set_local $duration)
...
```

Listing 10: Variant of btb_breakout with more instructions added in between time measurement.

WASM-MUTATE proves effective against cache access timers because the time measurement of single or a few instructions is inherently different. By introducing more instructions, this randomness is amplified, thereby reducing the timer's accuracy.

Furthermore, CPUs have a maximum capacity for the number of instructions they can cache. WASM-MUTATE injects instructions in such a way that the vulnerable instruction may exceed this cacheable instruction limit, meaning that caching becomes disabled. This kind of transformation can be viewed as padding [15]. In Listing 11 and Listing 12, we illustrate the effect of WASM-MUTATE on padding instructions. Listing 11 presents the original code used for training the branch predictor, along with the expected speculated code.

```
;; Code from original btb_breakout
...
;; train the code to jump here (index 1)
(i32.load (i32.const 2000))
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
(i32.load (i32.const 339968)) ;; S(83) is the secret
(i32.store (i32.const 83)) ;; just prevent optimization
```

Listing 11: Two jump locations in btb_breakout. The top one trains the branch predictor, the bottom one is the expected jump that exfiltrates the memory access.

```
;; Variant code
...
;; train the code to jump here (index 1)
<inserted instructions>
(i32.load (i32.const 2000))
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
;; transiently jump here
<inserted instructions>
(i32.load (i32.const 339968)) ;; "S"(83) is the secret
<inserted instructions>
(i32.store (i32.const 83)) ;; just prevent optimization
...
```

Listing 12: Variant of btb_breakout with more instructions added indindinctly between jump places.

The padding alters the arrangement of the binary code in memory, effectively impeding the attacker's capacity to initiate speculative execution. Even when an attack is launched and the vulnerable code is "speculated", the memory access is not impacted as planned.

In every program, we note that the exfiltration bandwidth tends to be greater than the original when the variants include a small number of transformations. This indicates that, although the transformations generally contribute to the reduction of data leakage, the initial few might not consistently contribute positively towards this objective. We have identified several fundamental reasons, which we discuss below.

Firstly, as emphasized in prior applications of WASM-MUTATE [8], uncontrolled diversification can be counterproductive if a specific objective, such as a cost function, is not established at the beginning of the diversification process. Secondly, while some transformations yield distinct WebAssembly binaries, their compilation produces identical machine code. Transformations that are not preserved undermine the effectiveness of diversification. For example, incorporating random `nop` operations directly into WebAssembly does not modify the final machine code as the `nop` operations are often removed by the compiler. The same phenomenon is observed with transformations to custom sections of WebAssembly binaries. Additionally, it is impor-

tant to note that transformed code doesn't always execute, i.e., WASM-MUTATE may generate dead code.

Finally, for ret2spec and pht, both programs are hardened with attack bandwidth reduction, but this does not materialize in a short-term timeframe (low count of stacked transformations). Furthermore, the exfiltration bandwidth is more dispersed for these two programs. Our analysis indicates a correlation between bandwidth reduction and the complexity of the binary subject to diversification. Ret2spec and pht are considerably larger than btb_breakout and btb_leakage. The former comprises more than 300k instructions, while the latter two include fewer than 800 instructions. Given that WASM-MUTATE applies precise, fine-grained transformations one at a time, the likelihood of impacting critical attack components, such as timing memory accesses, diminishes for larger binaries, particularly when limited to 1,000 transformations. Based on these observations, we believe that a greater number of stacked transformations would further contribute to eventually eliminating the attacks associated with ret2spec and pht.

Answer to RQ3: software diversification is effective at synthesizing WebAssembly binaries that are less susceptible to Spectre-like attacks. WASM-MUTATE generates variants of btb_breakout and btb_leakage that are totally protected and hardened variants of ret2spec and pht that are more resilient than the original program. In total, 70% of the diversified variants exhibit a reduced effectiveness (reduced data leakage bandwidth) compared to the original program. Larger programs require a greater number of transformations to effectively neutralize the attacks.

6. Discussion

Fuzzing WebAssembly compilers with WASM-MUTATE In fuzzing campaigns, selecting the appropriate starting inputs is both a significant challenge and essential for detecting bugs promptly [46]. This is particularly true with compilers, where the inputs should be well-formed yet intricate enough programs to probe various compiler components. WASM-MUTATE could address this challenge by generating semantically equivalent variants from an original WebAssembly binary, enhancing the scope and efficiency of the testing process. A practical example of this occurred in 2021, when this approach led to the discovery of a wasmtime security CVE [18]. Through the creation of semantically equivalent variants, the spill/reload component of cranelift was stressed, resulting in the discovering and subsequent resolution of the before-mentioned CVE.

Mitigating Port Contention Rokicki et al. [39] showed the practicality of a covert side-channel attack using port contention within WebAssembly code in the browser. This attack fundamentally relies on the precise prediction of Wasm instructions that trigger port contention on specific ports. To combat this security concern, we propose an man-in-the-middle mechanism, WASM-MUTATE, which could be conveniently implemented as a browser plugin. WASM-MUTATE has the ability to replace the WebAssembly instructions used as port contention predictor with other random instructions. This will inevitably remove the port contention in the specific port used to conduct the attack, hardening browsers against such exploitative maneuvers.

7. Related Work

Static software diversification refers to the process of synthesizing, and distributing unique but functionally equivalent programs to end users. The implementation of this process can take place at any stage of software development and deployment - from the inception of source code, through the compilation phase, to the execution of the final binary [24, 34]. WASM-MUTATE, a static diversifier, can be placed at the final stage, keeping in mind that the code will subsequently undergo final compilation by JIT compilers. The concept of software diversification owes much to the pioneering work of Cohen [12]. His suite of code transformations aimed to increase complexity and thereby enhance the difficulty of executing a successful attack against a broad user base [12]. WASM-MUTATE's rewriting rules draw significantly from Cohen and Forrest seminal contributions [12, 19].

Jackson and colleagues [24] proposed that the compiler can play a pivotal role in promoting static software diversification. In the context of WebAssembly, CROW leverages compiler technology for diversification. It is a superdiversifier [25],for WebAssembly that is built in the LLVM compilation tool chain. However, integrating the diversifier directly into the LLVM compiler, restricts the tool's applicability to WebAssembly binaries generated through LLVM. This implies that any WebAssembly source code that lacks an LLVM frontend implementation cannot take advantage of CROW's capabilities. In contrast, WASM-MUTATE provides a more versatile and faster WebAssembly to WebAssembly diversification solution, maintaining compatibility with any compiler. Secondly, unlike CROW, WASM-MUTATE does not rely on an SMT solver to validate the generated variants. Instead, it guarantees semantic equivalence by design, resulting in greater efficiency in generating WebAssembly variants, as discussed in subsection 5.1. As a WebAssembly to WebAssembly diversification tool, WASM-MUTATE augments the range of tools capable of generating WebAssembly programs, a topic explored comprehensively throughout this work.

The process of diversifying a WebAssembly program can be conceptualized as a three-stage procedure: parsing the program, transforming it, and finally re-encoding it back into WebAssembly. Our review of the literature has revealed several studies that have employed parsing and encoding components for WebAssembly binaries across various domains. This indicates that these works accept a WebAssembly binary as an input and output a unique WebAssembly binary. These domains span optimization [47], control flow [2], and dynamic analysis [31, 43, 2, 3]. When the transformation stage introduces randomized mutations to

the original program, the aforementioned tools could potentially be construed as diversifiers. WASM-MUTATE is related to these previous works, as it can serve as an optimizer or a test case reducer due to the incorporation of an e-graph at the heart of its diversification process [44]. To the best of our knowledge, the introduction of an e-graph into WASM-MUTATE marks the first endeavor to integrate an e-graph into a WebAssembly to WebAssembly analysis tool.

BREWasm [10] offers a comprehensive static binary rewriting framework for WebAssembly and can be considered to be the most similar to WASM-MUTATE. For instance, it can be used to model a diversification engine. It parses a Wasm binary into objects, rewrites them using fine-grained APIs, integrates these APIs to provide high-level ones, and re-encodes the updated objects back into a valid Wasm binary. The effectiveness and efficiency of BREWasm have been demonstrated through various Wasm applications and case studies on code obfuscation, software testing, program repair, and software optimization. The implementation of BREWasm follows a completely different technical approach. In comparison with our work, the authors pointed out that our tool employs lazy parsing of Wasm. Although they perceived this as a limitation, it is eagerly implemented to accelerate the generation of WebAssembly binaries. Additionally, our tool leverages the parser and encoder of wasmtime, a standalone compiler and interpreter for Wasm, thereby boosting its reliability and lowering its error-prone nature.

Another similar work to WASM-MUTATE is WASMixer [11]. WASMixer focuses on three code obfuscation methods for WebAssembly binaries: memory access encryption, control flow flattening, and the insertion of opaque predicates. Their strategy is specifically designed for obfuscating Wasm binaries. In contrast, while WASM-MUTATE does not employ memory access encryption or control flow flattening, it can still function effectively as an obfuscator. Previous evaluations confirm that WASM-MUTATE has been successful in evading malware detection [8]. On the same topic, Madvex [32] also aims to modify Wasm binaries to achieve malware evasion, but their approach is principally driven by a generic reward function and is largely confined to altering only the code section of a Wasm binary. WASM-MUTATE, however, adopts a more flexible strategy by applying a broader array of transformations, which are not limited to the code section. Consequently, WASM-MUTATE is capable of generating malware variants without negatively affecting either their code or performance.

8. Conclusion

WASM-MUTATE is a fast and effective diversification tool for WebAssembly, with a 100% diversification rate across the 303 programs of the considered benchmark. With respect to speed, it creates over 9000 unique variants per hour. The WASM-MUTATE workflow ensures that all final variants offer different and unique execution traces. We have proven that WASM-MUTATE is able to mitigate Spectre attacks in WebAssembly, producing fully protected variants of two versions of the btb attack, and variants of ret2spec and pht that leak less data than the original ones.

In future work, we aim to fine-tune the diversification process, balancing broad diversification with the needs of specific scenarios. Besides, the creation of rewriting rules for WASM-MUTATE is currently a manual task, yet we have identified potential for automation. For instance, WASM-MUTATE could be enhanced through data-driven methods such as rule mining. Furthermore, we have observed that the impact of WASM-MUTATE on ret2spec and pht attacks is considerably less compared to btb attacks. These attacks exploit the returning address of executed functions in the program stack. One mitigation of this would be multi-variant execution strategy, implemented on top of WASM-MUTATE. By offering different execution paths, the returning addresses on the stack at each function execution would vary, thereby improving the hardening of binaries against ret2spec attacks.

References

[1] Azad, B.A., Laperdrix, P., Nikiforakis, N., 2019. Less is more: Quantifying the security benefits of debloating web applications, in: 28th USENIX Security Symposium (USENIX Security 19), USENIX Association, Santa Clara, CA. pp. 1697–1714. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/azad.

[2] Breitfelder, F., Roth, T., Baumgärtner, L., Mezini, M., 2023. Wasma: A static webassembly analysis framework for everyone, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 753–757. doi:10.1109/SANER56733.2023.00085.

[3] Brito, T., Lopes, P., Santos, N., Santos, J.F., 2022. Wasmati: An efficient static vulnerability scanner for webassembly. Computers & Security 118, 102745. URL: https://www.sciencedirect.com/science/article/pii/S0167404822001407, doi:https://doi.org/10.1016/j.cose.2022.102745.

[4] Bruschi, D., Cavallaro, L., Lanzi, A., 2007. Diversified process replicæ for defeating memory error exploits, in: 2007 IEEE International Performance, Computing, and Communications Conference, pp. 434–441. doi:10.1109/PCCC.2007.358924.

[5] Cabrera-Arteaga, J., Donde, S., Gu, J., Floros, O., Satabin, L., Baudry, B., Monperrus, M., 2020. Superoptimization of webassembly bytecode, in: Proceedings of MoreVMs: Workshop on Modern Language Runtimes. URL: http://arxiv.org/pdf/2002.10213, doi:10.1145/3397537.3397567.

[6] Cabrera Arteaga, J., Laperdrix, P., Monperrus, M., Baudry, B., 2022. Multi-variant execution at the edge, in: Proceedings of the 9th ACM Workshop on Moving Target Defense, Association for Computing Machinery, New York, NY, USA. p. 11–22. URL: https://doi.org/10.1145/3560828.3564007, doi:10.1145/3560828.3564007.

[7] Cabrera Arteaga, J., Malivitsis, O.F., Pérez, O.L.V., Baudry, B., Monperrus, M., 2021. Crow: Code diversification for webassembly. URL: https://madweb.work/preprints/madweb21-paper4-pre_print_version.pdf, arXiv:2008.07185.

[8] Cabrera-Arteaga, J., Monperrus, M., Toady, T., Baudry, B., 2023. Webassembly diversification for malware evasion. Computers & Security 131, 103296. URL: https://www.sciencedirect.com/science/article/pii/S0167404823002067, doi:https://doi.org/10.1016/j.cose.2023.103296.

[9] Cao, D., Kunkel, R., Nandi, C., Willsey, M., Tatlock, Z., Polikarpova, N., 2023. Babble: Learning better abstractions with e-graphs and anti-unification. Proc. ACM Program. Lang. 7. URL: https://doi.org/10.1145/3571207, doi:10.1145/3571207.

[10] Cao, S., He, N., Guo, Y., Wang, H., 2023a. A General Static

Binary Rewriting Framework for WebAssembly. arXiv e-prints , arXiv:2305.01454doi:10.48550/arXiv.2305.01454, arXiv:2305.01454.

[11] Cao, S., He, N., Guo, Y., Wang, H., 2023b. WASMixer: Binary Obfuscation for WebAssembly. arXiv e-prints , arXiv:2308.03123doi:10.48550/arXiv.2308.03123, arXiv:2308.03123.

[12] Cohen, F.B., 1993. Operating system protection through program evolution. Computers & Security 12, 565–584.

[13] Crane, S.J., Volckaert, S., Schuster, F., Liebchen, C., Larsen, P., Davi, L., Sadeghi, A.R., Holz, T., De Sutter, B., Franz, M., 2015. It's a trap: Table randomization and protection against function-reuse attacks, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 243–255. URL: https://doi.org/10.1145/2810103.2813682, doi:10.1145/2810103.2813682.

[14] Dongarra, J.J., Hinds, A., 1979. Unrolling loops in fortran. Software: Practice and Experience 9, 219–226.

[15] Duck, G.J., Gao, X., Roychoudhury, A., 2020. Binary rewriting without control flow recovery, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 151–163. URL: https://doi.org/10.1145/3385412.3385972, doi:10.1145/3385412.3385972.

[16] D'Elia, D.C., Invidia, L., Palmaro, F., Querzoni, L., 2022. Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. Digital Threats 3. URL: https://doi.org/10.1145/3478520, doi:10.1145/3478520.

[17] Fastly, 2020. The power of serverless, 72 times over. URL: https://www.fastly.com/blog/the-power-of-serverless-at-the-edge.

[18] Fastly, 2021. Stop a wasm compiler bug before it becomes a problem | fastly. https://www.fastly.com/blog/defense-in-depth-stopping-a-wasm-compiler-bug-before-it-became-a-problem.

[19] Forrest, S., Somayaji, A., Ackley, D., 1997. Building diverse computer systems, in: Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133), pp. 67–72. doi:10.1109/HOTOS.1997.595185.

[20] Google, 2020. Safeside. https://github.com/PLSysSec/safeside. URL: https://github.com/PLSysSec/safeside.

[21] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017a. Bringing the web up to speed with WebAssembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 185–200.

[22] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J., 2017b. Bringing the web up to speed with webassembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 185–200. URL: https://doi.org/10.1145/3062341.3062363, doi:10.1145/3062341.3062363.

[23] Hilbig, A., Lehmann, D., Pradel, M., 2021. An empirical study of real-world webassembly binaries: Security, languages, use cases, in: Proceedings of the Web Conference 2021, pp. 2696–2708.

[24] Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., Franz, M., 2011. Compiler-generated software diversity, in: Moving Target Defense. Springer, pp. 77–98.

[25] Jacob, M., Jakubowski, M.H., Naldurg, P., Saw, C.W.N., Venkatesan, R., 2008. The superdiversifier: Peephole individualization for software protection, in: International Workshop on Security, Springer. pp. 100–120.

[26] Jetbrain, 2023. Kotlin wasm. https://kotlinlang.org/docs/wasm-overview.html. URL: https://kotlinlang.org/docs/wasm-overview.html.

[27] Kim, M., Jang, H., Shin, Y., 2022. Avengers, assemble! survey of webassembly security solutions, in: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), pp. 543–553. doi:10.1109/CLOUD55607.2022.00077.

[28] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y., 2019. Spectre attacks: Exploiting speculative execution, in: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19. doi:10.1109/SP.2019.00002.

[29] Koppel, J., Guo, Z., de Vries, E., Solar-Lezama, A., Polikarpova, N., 2022. Searching entangled program spaces. Proc. ACM Program. Lang. 6. URL: https://doi.org/10.1145/3547622, doi:10.1145/3547622.

[30] Le, V., Afshari, M., Su, Z., 2014. Compiler validation via equivalence modulo inputs, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 216–226. URL: https://doi.org/10.1145/2594291.2594334, doi:10.1145/2594291.2594334.

[31] Lehmann, D., Pradel, M., 2019. Wasabi: A framework for dynamically analyzing webassembly, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Association for Computing Machinery, New York, NY, USA. p. 1045–1058. URL: https://doi.org/10.1145/3297858.3304068, doi:10.1145/3297858.3304068.

[32] Loose, N., Mächtle, F., Pott, C., Bezsmertnyi, V., Eisenbarth, T., 2023. Madvex: Instrumentation-based Adversarial Attacks on Machine Learning Malware Detection. arXiv e-prints , arXiv:2305.02559doi:10.48550/arXiv.2305.02559, arXiv:2305.02559.

[33] Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K., 2005. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices 40, 190–200.

[34] Lundquist, G.R., Mohan, V., Hamlen, K.W., 2016. Searching for software diversity: attaining artificial diversity through program synthesis, in: Proceedings of the 2016 New Security Paradigms Workshop, pp. 80–91.

[35] Nandi, C., Willsey, M., Anderson, A., Wilcox, J.R., Darulova, E., Grossman, D., Tatlock, Z., 2020. Synthesizing structured cad models with equality saturation and inverse transformations, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 31–44. URL: https://doi.org/10.1145/3385412.3386012, doi:10.1145/3385412.3386012.

[36] Narayan, S., Disselkoen, C., Moghimi, D., Cauligi, S., Johnson, E., Gang, Z., Vahldiek-Oberwagner, A., Sahita, R., Shacham, H., Tullsen, D., Stefan, D., 2021. Swivel: Hardening WebAssembly against spectre, in: 30th USENIX Security Symposium (USENIX Security 21), USENIX Association. pp. 1433–1450. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[37] Premtoon, V., Koppel, J., Solar-Lezama, A., 2020. Semantic code search via equational reasoning, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 1066–1082. URL: https://doi.org/10.1145/3385412.3386001, doi:10.1145/3385412.3386001.

[38] Ren, X., Ho, M., Ming, J., Lei, Y., Li, L., 2021. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study, in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 142–157. URL: https://doi.org/10.1145/3453483.3454035, doi:10.1145/3453483.3454035.

[39] Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y., 2022. Port contention goes portable: Port contention side channels in web browsers, in: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA. p. 1182–1194. URL: https://doi.org/10.1145/3488932.3517411, doi:10.1145/3488932.3517411.

[40] Rossberg, A., 2019. WebAssembly Core Specification. Technical Report. W3C. URL: https://www.w3.org/TR/wasm-core-1/.

[41] Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Lup, G., Taneja, J., Regehr, J., 2017. Souper: A Synthesizing Superopti-

mizer. arXiv e-prints , arXiv:1711.04422doi:10.48550/arXiv.1711.04422, arXiv:1711.04422.

[42] Schwarz, M., Maurice, C., Gruss, D., Mangard, S., 2017. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript, in: Kiayias, A. (Ed.), Financial Cryptography and Data Security, Springer International Publishing, Cham. pp. 247–267.

[43] Stiévenart, Q., De Roover, C., 2020. Compositional information flow analysis for webassembly programs, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE. pp. 13–24.

[44] Tate, R., Stepp, M., Tatlock, Z., Lerner, S., 2009. Equality saturation: A new approach to optimization, in: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, NY, USA. p. 264–276. URL: https://doi.org/10.1145/1480881.1480915, doi:10.1145/1480881.1480915.

[45] Wagner, L., Mayer, M., Marino, A., Soldani Nezhad, A., Zwaan, H., Malavolta, I., 2023. On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot, in: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 72–82. URL: https://doi.org/10.1145/3593434.3593454, doi:10.1145/3593434.3593454.

[46] Wang, J., Chen, B., Wei, L., Liu, Y., 2017. Skyfire: Data-driven seed generation for fuzzing, in: 2017 IEEE Symposium on Security and Privacy (SP), pp. 579–594. doi:10.1109/SP.2017.23.

[47] Wen, E., Dietrich, J., 2023. Wasmslim: Optimizing webassembly binary distribution via automatic module splitting, in: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 673–677. doi:10.1109/SANER56733.2023.00069.

[48] Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P., 2021. Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. 5. URL: https://doi.org/10.1145/3434304, doi:10.1145/3434304.