WasmDyPa: Effectively Detecting WebAssembly Bugs via Dynamic Program Analysis

Wenlong Zheng Baojian Hua* Zhuochen Jiang
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
{zwl21, jzc666}@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—Safe binary execution is often a crucial requirement in today's security critical computing infrastructures. WebAssembly is an emerging language designed for safe binary execution that has been deployed in many security critical domains, such as blockchain, edge computing, and clouds. However, WebAssembly's security guarantee is not a panacea, and recent studies have revealed a large spectrum of security issues such as integer overflows and memory vulnerabilities, leading to serious security hazards to WebAssembly applications.**

In this paper, we propose the first automated bug detection framework for WebAssembly programs based on dynamic program analysis, directly on WebAssembly binaries. To realize the whole process, we present WasmDyPa, the first dynamic bug detection system, consisting of three primary components: 1) a tainted input generation for WebAssembly binaries; 2) a static instrumentation hook providing extensible interfaces to collect and record runtime information; and 3) dynamic program analysis algorithms as security plugins to detect vulnerabilities. We have implemented a software prototype for WasmDyPa, and have conducted experiments to evaluate the effectiveness, usefulness, performance and overhead of our approach. The experimental results demonstrated that WasmDyPa can accurately detect vulnerabilities with a 88.24% precision and a 93.75% recall. Furthermore, WasmDyPa detected 56 bugs in real-world WebAssembly programs, including 2 integer overflows and 54 memory bugs.

***Index Terms*—WebAssembly, Security, Dynamic analysis**

I. Introduction

Today's cloud or edge computing infrastructures put forward higher requirements for the safe execution of binary programs on them. For example, in a multi-tenant scenario, inter-process separation without sacrificing execution is essential to isolate different tenants [1]. WebAssembly (Wasm) [2] is an emerging portable instruction set architecture and bytecode distribution format that allows for safe program execution with near-native execution efficiency. Due to Wasm's technical advantages of type safety [3] and intra-process lightweight sandboxing [4], Wasm has been extensively used in a large spectrum of safety-critical scenarios such as cryptography [5], smart contracts [6], cloud computing [7], embedded devices [8], and Internet-of-Things [9].

While Wasm makes a significant step toward defining a secure binary distribution format, existing studies [10] [11] have revealed that Wasm programs are still vulnerable and exploitable due to two main root causes: 1) type system defects; and 2) linear memory overflows. First, Wasm, starting from its initial design, incorporated a strong type system [3] with a mathematically rigorous type safety proof. While Wasm's type system is essential in guaranteeing type safety, it cannot guarantee arbitrary safe properties due to its specific design defects. For example, Wasm does not check integer overflows by tracing value propagations in programs, leading to generations of potential undetected overflows. Worse yet, such undetected overflows might lead to buffer overflows, when being used in memory allocations [12]. Despite this urgent needs, existing studies and tools [13] [14] [15] for Wasm cannot detect these vulnerabilities caused by type system design defects.

Second, to guarantee control flow integrity [16], Wasm introduced a fine-grained memory model [17] to store function return address and function data in separate stacks called *managed memory* and *linear memory*, respectively. While Wasm's linear memory mitigates return-oriented programming (ROP) [18] based attacks effectively, existing studies [10] [11], unfortunately, have revealed that Wasm programs are still vulnerable and exploitable. Worse yet, Wasm does not support garbage collections but relies on manual memory management, which might lead to the notorious memory vulnerabilities such as double-free (DF) [19] or use-after-free (UAF) [20]. Despite the fact that current studies have empirically studied Wasm compilers [21] or runtimes bugs [22], as well as mitigations (*e.g.*, stack canary [23]), a systematic study of effective bug detection for Wasm is still missing.

Recognizing this security criticality and urgency, we propose using a dynamic analysis approach to detect potential vulnerabilities by leveraging runtime information. Specifically, we argue that making a dynamic analysis framework dedicated to Wasm with the aid of static instrumentation has the following advantages: 1) *precision*: dynamic analysis provides more precise bug identification information by leveraging accurate runtime information, which is often missing in a pure static analysis; 2) *effectiveness*: dynamic analysis is effective in collecting and tracking context-sensitive information such as the order of memory accesses and the traces of function invocations, which is often difficult or even impossible for a pure static analysis; and 3) *efficiency*: dynamic analysis is often efficient in that it only affects the susceptible Wasm instructions via a selective static instrumentation before the dynamic analysis. It is notable that our approach can be

regarded as a dynamic enhancement to Wasm's already strong static (*e.g.*, type system [24]) or dynamic (*e.g.*, sandboxing [25]) security mechanisms, and thus supplements them.

Following these insights, in this work, we present WASMDYPA, a fully automated bug detection framework for Wasm programs based on dynamic analysis in conjunction with static instrumentation, leverages runtime information that is hard to obtain in static analysis. WASMDYPA consists of three main components: 1) a Wasm compatible tainted test case generation, to generate tainted test cases covering sink sites; 2) static instrumentation hooks to track and collect runtime information of susceptible Wasm instructions; and 3) dedicated analysis algorithms as program analysis plugins to detect security vulnerabilities, by analyzing available runtime information.

To realize the whole process, we have implemented a software prototype for WASMDYPA. First, we implemented a tainted test generation by utilizing static binary instrumentation techniques to emulate a tainted input by imposing restrictions on the potential sink sites. Furthermore, to improve the path coverage, we leveraged and extended the existing path-coverage oriented Wasm fuzzer. Second, we implemented the instrumentation of hooks by a dedicated compiler rewriting pass to traverse the Wasm abstract syntax tree data structures. Moreover, we utilized a validator to check the Wasm program's semantic consistency before and after the instrumentation. Third, we implemented diverse dynamic analysis algorithms as security plugins to detect potential vulnerabilities, based on the runtime data collected by instrumented hooks.

To validate our design and implementation, we conducted a systematic evaluation of WASMDYPA, in terms of its effectiveness, performance, and usefulness on both micro- and real-world benchmarks. First, we evaluated WASMDYPA on a microbenchmark WasmBench with ground truth, and WASMDYPA is effective in achieving a 88.24% precision and a 93.75% recall. Second, WASMDYPA is efficient in processing Wasm programs with acceptable overhead. Third, WASMDYPA is useful to detect real-world bugs: from a benchmark with 941 Wasm programs, WASMDYPA successfully detected 56 bugs, including 2 integer overflows, and 54 memory bugs.

Contributions. To the best of our knowledge, this work is the *first* systematic study of effective Wasm bug detection via dynamic program analysis. To summarize, our work makes the following contributions:

- **Infrastructure design.** We designed the first framework, WASMDYPA, to effectively detect Wasm bugs via dynamic program analysis.
- **Prototype implementation.** We have implemented a software prototype to validate our system design.
- **Extensive evaluation.** We have conducted extensive experiments to evaluate the effectiveness, performance, and usefulness of WASMDYPA on microbenchmarks as well as real-world Wasm applications.
- **Open source.** We make our software prototype, datasets, and evaluation results publicly available in the interest of open science at: https://doi.org/10.5281/zenodo.8012460.

The rest of this paper is organized as follows: Section II presents the background on Wasm and dynamic analysis. Section III presents the motivation and threat model for this work. Sections IV and V present the design and implementation of WASMDYPA, respectively. Section VI presents the experiments we performed to evaluate WASMDYPA. Section VII discusses limitations and directions for future work. Section VIII discusses the related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work: background on WebAssembly (§ II-A), and dynamic analysis (§ II-B).

A. Wasm

Brief history. Wasm was initially proposed by Google and Mozilla in 2015 [26] and became a de facto standard language in browsers in 2017 [27]. With the first complete formal definition of Wasm released in 2018 [28], Wasm was announced as an official (and the fourth) Web standard by the W3C [29] in 2019. With the design of the WebAssembly System Interface (WASI) [30] and the standard version 2.0 draft of Wasm [28], Wasm has grown into a stable and production-quality language to be used in Web and standalone domains.

Feature. Wasm emphasizes safety, efficiency, and portability [31]. First, to ensure program safety, Wasm incorporates secure features such as strong typing, sandboxing isolation and secure control flow [3] [25]. Second, Wasm's virtual machine (VM) balances space usage and execution efficiency, enabling it to fully utilize hardware capabilities on diverse platforms with high efficiency. Third, WASI provides safe system interaction interfaces, making it possible to deploy programs out of browsers.

Applications. Wasm's advanced features have led to its wide adoption in both Web and non-Web domains. In Web domains, Wasm is the fourth official language (after HTML, CSS, and JavaScript) with full support from major browsers. In non-Web domains, Wasm has been used in diverse computing scenarios such as cloud computing [32] [33], IoT [34], blockchain [6] [35] [36] [37], edge computing [38], video transcoder [39], and game engines [40]. In the future, a desire to secure the cloud or edge computing infrastructures without sacrificing efficiency will make Wasm a more promising language.

B. Dynamic Program Analysis

Dynamic analysis [41] is a well-established technique to analyze properties of softwares or systems by observing their runtime behavior. Unlike static analysis [42], which examines the source code of an application without executing it, dynamic analysis examines how an application behaves in real-world scenarios and has been recognized as an effective technique complementing static analysis.

Dynamic analysis analyzes all possible runtime properties of a target program, including but not limited to the program's execution trace, input/output behavior, and memory usage. The analysis results can be further used to investigate and identify

diverse program defects such as concurrency bugs [43] [44], memeory bugs [45], and performance issues [46].

Due to its technical advantages of precision, effectiveness, and versatility, dynamic analysis has been used to analyze a wide spectrum of languages C/C++ [47], JavaScript [48] and even x86 binaries [49].

III. Motivation and Threat Model

In this section, we first present an overview of Wasm security (§ III-A) and motivating examples (§ III-B), then give the threat model (§ III-C) for this work.

A. Wasm Security Overview

Despite Wasm's design goal of security, vulnerabilities still exist in real-world Wasm programs [10]. By carefully inspecting the existing vulnerabilities, we focus on two categories of Wasm bugs manifesting in in-the-wild Wasm programs [11]: integer overflows, and memory corruptions.

Integer overflows. Although Wasm's strong type system guarantees type safety [3] [50], it does not check integer overflows by tracking data propagations, leading to potential integer overflows. Worse yet, such an integer overflow may further lead to buffer overflows (IO2BO) [12], when the integer is used in memory allocation. Despite these urgent security needs, existing studies and tools [13] [14] [15] for Wasm cannot detect vulnerabilities caused by integer overflows.

Memory corruptions. Although the Wasm memory module isolates managed data and function return address to prevent return-oriented progmmaing (ROP) based attacks [18], an attacker can still corrupt the memory, in two ways: first, an attacker can compromise data stored on the unmanaged stack by triggering buffer overflows, which may overwrite not only local variables in the same stack, but also other stack frames upward in the unmanaged stack. Second, an attacker may corrupt memory in Wasm programs by tampering with the heap metadata of the memory allocator. While standard memory allocators (*e.g.*, `dlmalloc` [51]) have been hardened against a variety of metadata corruption attacks, such attacks are still feasible on metadata (*e.g.*, the classical unlink exploit [10]). Hence, a systematic and effective approach is essential to detect such vulnerabilities.

B. Motivating Examples

To put the above discussion of Wasm bugs in perspective, we present, in Fig. 1 and Fig. 2, two sample Wasm programs containing an integer overflow (leading to IO2BO) and a double-xfree memory issue we adapted from real-world vulnerabilities CWE-190 [52] and CWE-415 [19], respectively.

IO2BO. An IO2BO bug occurs when an overflowed value is used in memory allocation, leading to subsequent buffer overflows. Fig. 1 gives a Wasm IO2BO bug we adapted from a real-world CWE [52] in OpenSSH 3.3 [53]. This code snippet allocates memory for a `char*` array to store network packets. However, if the number of packets `n` is large enough (line 1), the integer multiplication result `n<<2` (hence the allocated memory size) would overflow (line 3). For example, for $n =$ `0xC0000000`, we have $n << 2 = 0$. As a result, any subsequent buffer accesses will lead to overflows on this zero-length buffer.

```
1 local.get 5 // get number of packet from the variable 5
2 i32.const 2 // store 2 on the top of the stack
3 i32.shl // (the number of packet) * (sizeof(char*))
4 call malloc // potential sink site
```

Fig. 1: A sample Wasm program illustrating IO2BO bugs we adapted from a real-world vulnerability CWE-190 [52].

```
1 local.get 2 // get the base address from variable 2
2 i32.load offset=8 // load data from base address + offset
3 local.set 25 // assign variable 25 with data from stack
4 local.get 25 // place value of variable 25 at stack top
5 call free // call free() with a pointer on stack
6 ...
7 // omitted due to the similarity
8 local.get 2
9 i32.load offset=8
10 local.set 26
11 local.get 26
12 call free // call free() again with a freed pointer
```

Fig. 2: A sample Wasm program illustrating double-free bugs we adapted from a real-world vulnerability CWE-415 [19].

While static analysis has difficulty detecting this bug precisely due to the lack of concrete value for `n`, dynamic analysis is able to catch such bugs, by generating specific input for `n` to trigger the overflow. Furthermore, as dynamic analysis keeps track of the propagation of the value which sinks into a memory allocation site (*i.e.*, `malloc` in this example), it can generate informative diagnosis for subsequent debugging or analysis, a valuable capability for end developers.

Double-free. A double-free bug manifests when an already released memory is freed for a second time. In Fig. 2, the variable identified by index 2 stores the base address of the memory to be accessed, which is first placed on top of the operand stack (line 1). Next, the memory pointed by this address is first released by the function `free` (line 5), then is released again (line 12), leading to a double-free bug.

Dynamic analysis can detect memory corruptions like double-free effectively. Specifically, for double-free bugs, by recording and keeping track of already released memory addresses, dynamic analysis can determine effectively and efficiently whether or not a pointer is valid and thus releasable.

C. Threat Model

Wasm has a rich ecosystem consisting of high-level language support, compilation toolchains, binary representation, and Wasm VM. The focus of this work is on dynamic vulnerability detection based on Wasm binary representation for any underlying Wasm VM. Therefore, we make the following assumptions in the threat model of this work.

We assume the compilation toolchains are trustworthy in producing semantics equivalent Wasm programs from sources. On the one hand, with considerable efforts in development and testing, compiler toolchains for Wasm are becoming mature

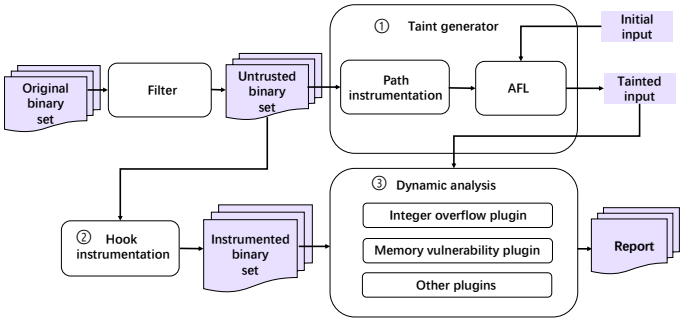


Fig. 3: The architecture of WASMDYPA.

[54]. On the other hand, many studies are devoted to testing or fuzzing compilers to detect potential bugs [55] [56] [57] [58]. Hence, this assumption is reasonable in practice.

We assume the underlying Wasm VM executing Wasm programs is trustworthy. On the one hand, extensive security studies have been conducted on Wasm VM (*e.g.*, empirical studies [22], and fuzzing [59]). On the other hand, Wasm VM security studies are orthogonal to and supplement the security study of Wasm programs in this work.

We assume Wasm programs may contain bugs and thus are not trustworthy. Such bugs are introduced either by specific design defects of Wasm (e.g., lack of integer overflow detection) or by vulnerabilities in insecure source language such as C/C++ [60]. Hence, the study of bug detection in this work is essential to guarantee and enhance security of Wasm programs.

IV. WASMDYPA DESIGN

In this section, we present the design of WASMDYPA. We first describe its design goals (§ IV-A), overall architecture (§ IV-B), and language model (§ IV-C). We then discuss the tainted input generation (§ IV-D), the hook instrumentation (§ IV-E), and the dynamic detection plugins (§ IV-F).

A. Design Goals

We design WASMDYPA with three goals: 1) high detection accuracy, 2) full automation and easy extension, and 3) low overhead. First, WASMDYPA should detect potential Wasm bugs accurately, by leveraging runtime information generated by actual program execution. Second, WASMDYPA should be fully automated to minimize manual intervention and manual efforts. Furthermore, WASMDYPA should be extensible in processing bugs besides integer overflows and memory corruptions studied in this work. Third, WASMDYPA should achieve low overhead in terms of the instrumentation time, execution time, and analysis time.

B. Overall Architecture

We present, in Fig. 3, the overall architecture of WASMDYPA, consisting of three primary modules: 1) a tainted input generator (①); 2) a hook instrumentation (②); and 3) dynamic analysis plugins (③). First, the tainted input generator module aims to trigger potential vulnerabilities in the target Wasm, by generating effective tainted inputs. Second, the hook instrumentation module takes a Wasm program as input and rewrites the program by placing hooks around susceptible Wasm instructions to collect runtime data for further dynamic analysis. Third, the dynamic security plugins retrieve and analyze runtime data to detect potential vulnerabilities and bugs.

Val. Type	ρ	::=	`i32` | `i64` | `f32` | `f64`
Func. Type	σ	::=	$\rho^* \to \rho^*$
Type	τ	::=	ρ | σ
Binary Op.	b	::=	`i32.add` | `i32.mul` | `i32.shl` | ...
Unary Op.	u	::=	`i32.abs` | `i32.eqz` | ...
Load/Store	l	::=	ρ`.load` | ρ`.store`
Local Op.	c	::=	`local.(set` | `get` | `tee)` x
Global Op.	g	::=	`global.(set` | `get)` x
Call	t	::=	`call` f | `call_indirect` σ
Instr.	i	::=	b | u | l | c | g | t
		|	`drop` | `nop` | `if` | `else` | `block`
		|	`loop` | `end` | `br` a | `br_if` a
		|	`br_table` a^+ | `select`
		|	`memory.grow` | ρ`.const` c | ...
Function	f	:=	$\sigma\ x\{i^*\}$
Module	m	:=	f^*

Fig. 4: Core syntax of Wasm language.

C. Language Model

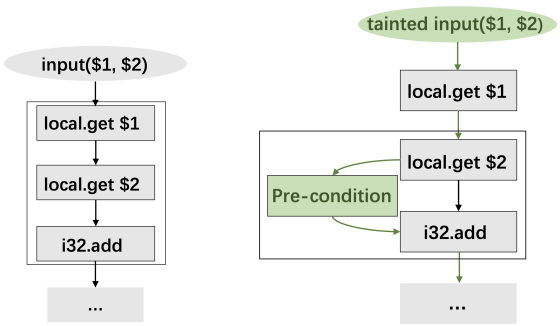
We introduce a simple language model capturing the core syntax of Wasm, to introduce both the tainted input generation and hook instrumentation. We present, in Fig. 4, the core syntax of Wasm via a context-free grammar.

Each Wasm module m consists of a list of functions f, whose body contains a sequence of instructions i. A function f may have multiple arguments and return results, indicated by its type $\rho^* \to \rho^*$ (the notation * stands for a Kleene closure).

An instruction i consists of binary/unary operations, memory load/stores, structured control flows, and function invocation/return, with some irrelevant instructions omitted for brevity. Wasm instructions demonstrated three distinct properties: first, Wasm is a stack-based VM in that operands and the result of an operation are always on top of the operand stack. For example, the addition operation `i32.add` pops two operands from stack and pushes the result. Second, Wasm instructions are strongly typed in specifying the expected type in the opcode (*e.g.*, the `i32` in `i32.abs`), facilitating binary-level type checking. Third, Wasm supports *structured* control flows (*e.g.*, `if` or `loop`), making compilation to Wasm easier.

D. Tainted Input Generation

Effective dynamic detection of potential vulnerabilities often requires the availability of tainted input data [49], whose propagation can trigger deep bugs. Hence, to detect Wasm bugs dynamically, a well-designed tainted input generator is



(a) Control flow before instrumentation (b) Control flow after instrumentation

Fig. 5: The execution path before and after pre-condition instrumentation.

essential. However, current Wasm fuzzing studies and tools for Wasm (*e.g.*, WAFL [61], or Fuzzm [23]) focus on the coverage of execution paths, but lack the capability of tainted input generation. To address this problem, we propose an approach of *pre-condition instrumentation* to generate tainted inputs by instrumenting security predicates as pre-conditions.

Pre-condition instrumentation. The pre-condition instrumentation takes as input a Wasm program, and instruments it by imposing pre-conditions on target operations. To better illustrate the key insight, we present, in Fig. 5, a sample Wasm program. To cover a path, the existing fuzzing-based approach would generate a pair of input (*e.g.*, $1 = 10, $2 = 12). However, while this input covered the execution path, it failed to trigger the the potential overflows for the `i32.add` instruction (Fig. 5a). To tackle this problem, we instrument the target instruction with a pre-condition. For this addition instruction, we instrument this pre-condition $(\$1 > 0 \wedge \$2 > 0 \wedge \$1 + \$2 < 0) \vee (\$1 < 0 \wedge \$2 < 0 \wedge \$1 + \$2 > 0)$, before `i32.add`, to guide the generation of new inputs satisfying this condition (Fig. 5b). With this instrumentation, a new pair of input triggering the overflow (*e.g.*, $1 = `0xFFFFFFFF` and $2 = 1), will be generated.

With this key insight, we present, in TABLE I, the representative overflow predicates to be used as pre-conditions instrumented into the target Wasm programs. (Note that the above illustrating example just made use of the predicate in the first row of the table.)

TABLE I: Overflow predicates used as pre-conditions.

Arithmetic operations	Overflow predicates
$x = o_1 +_s o_2$	$(o_1 > 0 \wedge o_2 > 0 \wedge x < 0) \vee (o_1 < 0 \wedge o_2 < 0 \wedge x > 0)$
$x = o_1 -_s o_2$	$(o_1 > 0 \wedge o_2 < 0 \wedge x < 0) \vee (o_1 < 0 \wedge o_2 > 0 \wedge x > 0)$
$x = o_1 *_s o_2$	$x \neq 0 \wedge x/o_1 \neq o_2$
$x = o_1 \ll o_2$	$x \gg o_2 \neq o_1$

E. Hook Instrumentation

The hook instrumentation takes as input a Wasm program and outputs the instrumented Wasm by placing hooks before and after corresponding instructions, to collect necessary runtime information for subsequent analysis.

We proposed a hook instrumentation strategy in a syntax-directed manner following the instruction i in Fig. 4. For the purpose of this paper, we present detailed designs of two specific hooks next: integer hooks and memory hooks.

Integer hooks. Integer hooks track the inputs and output of binary operations. We follow two steps to detect integer overflows: first, we instrument a `before_binary` hook and a `after_binary` hook, before and after a `binary` operation, to record its operands and result, respectively. Second, we identify an integer overflow by analyzing operands and result, based on the overflow predicates.

To accurately detect memory issues caused by integer overflows (*e.g.*, IO2BO), we need to track the propagation of integer values to identify sink sites where an overflowed value is used in memory allocation. To perform such value tracking efficiently, we follow the following two steps: first, upon detecting an integer overflow, we do not stop the dynamic analysis. Instead, we issue an error of integer overflow, then replace the overflowed value with a *dirty* value from a *dirty-zone*, following prior work [62]. Technically, the *dirty-zone* represents a value range of rarely used numbers, with the desired property of dirty stability, that is, the result of dirty operands remains dirty with high probability. In this work, we have selected a *dirty-zone* [`0xA0000000, 0xE0000000`) and an initial *dirty* value `0xC0000000`, which gives a probability of 99.99776% under a sequence of 10 arithmetic operations. Second, we resume the dynamic analysis with the *dirty* values propagated. For memory allocations, we check their argument of memory size and report a potential IO2BO vulnerability when the size is a *dirty* value.

Memory hooks. Memory hooks record the address of each memory along with its size, for either stack or heap allocation.

To identify memory vulnerabilities, we follow three steps: first, we instrument the hook `before_alloc` and `after_alloc` before and after a memory allocation, to collect the memory allocation size and the allocated address, respectively. Second, we instrument hooks before each memory operation to record the memory address in the operation. Third, we identify memory vulnerabilities with a syntax-directed approach, following prior work [63]. Specifically, to detect a double-free (DF), we check whether the memory address has already been released; and to detect a buffer overflow, we check whether the memory address exceeds the buffer boundary.

F. Dynamic Analysis Algorithm as Security Plugins

We designed an architecture for WASMDYPA of security plugins to detect potential vulnerabilities. Next, present two concrete security plugins as dynamic analysis algorithms.

Integer overflows. We present, in Algorithm 1, how an arithmetic overflow and IO2BO are detected, with the key idea of *dirty* value propagation as we have just discussed.

Algorithm 1: Integer overflow detection algorithm.

Input: $\mathbb{I}$: Wasm instructions, S: the operation stack
Output: S': the updated stack

```
1  Function arithOFDetector(𝕀, S):
2      𝕋 ← nextInstruction (𝕀);
3      if 𝕋 == arith then
4          isOF ← checkOverflow(S);
5          if isOF then
6              report(intOF);
7              S' ← stackUpdate(S, dirtyValue);
8          else
9              S' ← stackResume(S, origValue);
10     else
11         S' ← stackResume(S, origValue);
12     return S';
13 Function IO2BODetector(𝕀, S):
14     𝕋 ← nextInstruction (𝕀);
15     if 𝕋 == memAlloc then
16         isDirty ← checkDirtyVal(S);
17         if isDirty then
18             report(IO2BO);
19         else
20     else
21     S' ← stackResume(S);
22     return S';
```

Algorithm 2: Memory vulnerability detection.

Input: B: the address of the current memory base, K: a list to record memory information; others are same as in Algorithm 1

```
1  Function MemVulDetector(𝕀, S, B, K):
2      𝕋 ← nextInstruction (𝕀);
3      if 𝕋 == memAlloc then
4          size ← getStack(S);
5          p ← B;
6          B ← B + size;
7          K ← recordMemInfo(size, p, alloc)
8      else if 𝕋 == memFree then
9          size ← getStack(S);
10         B ← B - size;
11         p ← B;
12         K ← recordMemInfo(size, p, free)
13     else if 𝕋 == memAccess then
14         Addr ← getAddress(S);
15         isLegal ← validateAddress(K, Addr);
16         if isLegal then
17             continue;
18         else
19             report(MemVulnerability);
20     else
21         return;
```

To realize this key idea, the function `arithOFDetector` takes as inputs a list of Wasm instructions $\mathbb{I}$ and the current operation stack S, and returns the new operation stack S' as the result of executing the next instruction $\mathbb{T}$ in $\mathbb{I}$. This function consists of three key steps: first, the function analyzes the next instruction $\mathbb{T}$ to determine whether it is an arithmetic instruction by referring to the Wasm syntax in Fig. 4. Second, for the arithmetic operation, the algorithm checks whether or not an overflow manifests, by inspecting the operation stack S (line 4). The algorithm will report an error of overflow, before updating the operation stack with a *dirty* value and resuming the dynamic analysis (line 5 to 7). Third, for normal operations without overflows, the dynamic analysis continues with original values to keep the operand stack balanced.

Similarly, the function `IO2BODetector` also takes as inputs a list of Wasm instructions $\mathbb{I}$ and the current operation stack S, and returns the new operation stack S' as the result of executing the next instruction $\mathbb{T}$. During the analysis, the function determines potential IO2BO vulnerability by identifying *dirty* values on the operand stack S with respect to the specified *dirty-zone* .

Memory vulnerability. We present, in Algorithm 2, how memory vulnerabilities are detected.

The function `MemVulDetector` takes as inputs the B to record the address of memory base, and the list K to record memory information, others are same as in Algorithm 1. This function consists of 2 steps: first, the function updates and records every memory behavior (*i.e.*, stack frame arrangement and heap allocation) information (line 3 to 12). It is notable that the function differentiates the memory allocation (line 3 to 7) and release (line 8 to 12). Second, the function validates every memory access and outputs diagnosis in terms of memory vulnerability by referring to the list K (line 13 to 19). To be more specific, the function detects a double-free by comparing the address to be freed with the already released memory address recorded in K (line 12); the function detects use-after-free by comparing current memory access address with the released memory address, similar to double-free, and informs a use-after-free when program attempts to access the released memory; and the function detects the buffer overflow by comparing the address with the allocated memory space (*i.e.*, $[p, p + size]$, the base and boundary information for each memory piece), and the address exceeding the memory boundary indicates a buffer overflow.

V. WasmDyPa Implementation

To validate the system design, we have implemented a software prototype for WasmDyPa, consisting of the three aforementioned components in § IV. The prototype is distributed in our open source.

Tainted input generation. We have implemented the tainted input generation as a compiler rewriting pass to instrument overflow predicates, by leveraging the abstract syntax trees offered by Fuzzm [23]. Furthermore, to drive the whole

process of dynamic generation with the instrumented Wasm program, we have leveraged the fuzzing module of Fuzzm, which is in turn ported from AFL [64].

Hook instrumentation. We have implemented the hook instrumentation by porting and extending a popular hook infrastructure Wasabi [14], to collect and record runtime information. The porting and extension is nontrivial, during which we have addressed two key technical challenges: first, the initial Wasabi was designed for Web scenario lacking the capability to execute arbitrary non-Web Wasm programs. Hence, we ported it to Wasm-Micro-Runtime (WAMR) [65], a standalone Wasm VM. We select WAMR as our VM for 2 reasons: 1) WAMR is a popular Wasm VM with 3.8k Github stars, a criterion use in prior work [66]; and 2) WAMR is designed with a small footprint, high performance, and highly configurable features. We then implemented our hook instrumentation algorithms as C libraries following the native function interface standard provided by WAMR. Second, the initial hooks supplied by Wasabi are implemented in JavaScript and are thus untyped. We have implemented them in C, enabling extra static type checking to guarantee the type safety. Furthermore, we have implemented a validator for the hooks by leveraging `wasm-validate` facility from the WABT [67], offering well-formedness guarantees and consistency of instrumented Wasm code.

Security plugins. We have implemented the security plugins as customized dynamic analysis algorithms, which now consist of 855 lines of C code as native import functions for Wasm. To trace data propagation, we defined a set of global variables to represent the *dirty* value and *dirty-zone*. To record the memory information, we utilized hash tables to record memory base and its corresponding size. For memory allocation, we inserted its address and size into the table, and for memory release, we removed the address from the table. Finally, to give an informative diagnosis, we logged the operation details to facilitate root causes analysis of vulnerabilities.

VI. Evaluation

In this section, we present experiments to evaluate WasmDyPa. We first present the research questions guiding the experiments (§ VI-A), the benchmark we created (§ VI-C), and the experimental results in terms of the effectiveness, performance and overhead, and usefulness (§ VI-D to § VI-F). We then present the case study of real-world vulnerabilities WasmDyPa detected (§ VI-G).

A. Research Questions

By presenting the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. As WasmDyPa is proposed for bug detection, is it effective in detecting bugs in Wasm programs?

RQ2: Performance and overhead. As WasmDyPa makes of static instrumentation as well as dynamic analysis to detect vulnerabilities, what is its performance and overhead?

RQ3: Usefulness. Is WasmDyPa useful in detecting real-world bugs in real-world Wasm applications?

B. Experimental Setup

All the experiments and measurements are performed on a server with one 8 physical Intel i5 core (8 hyper thread) CPU and 8 GB of RAM running Ubuntu 20.04.

C. Datasets

We used two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world benchmarks with 941 Wasm programs (61 collected from CWE, and 880 Wasm binaries). The prototype is distributed in our open source.

Micro-benchmark. Evaluating the effectiveness of WasmDyPa needs a benchmark suite that comes with ground truth for analysis. Yet such a benchmark suite is not available (to the best of our knowledge) while curating the ground truth for large/complex, real-world programs may not be feasible. We thus took the first step to manually create **WasmBench**, a micro benchmark for vulnerability program analysis. As shown in TABLE II, we manually constructed a microbenchmark of 20 test cases with both normal programs and vulnerable programs with diverse types of vulnerabilities (as presented by the 2nd column), including integer overflows, double-free (DF), use-afer-free (UAF), and buffer overflow.

Currently, WasmBench only includes integer overflows and memory vulnerabilities. We will maintain and augment it by including more benchmarks and test cases while covering other vulnerabilities.

Real-world Wasm programs. Our real-world Wasm dataset consists of 941 unique Wasm binaries in the wild, in which 880 programs are created by referring to the dataset in prior work [11], and 61 are collected and compiled from CWE [68]. Evaluating WasmDyPa on this macro benchmark demonstrates the effectiveness and usefulness of WasmDyPa on real-world Wasm applications from diverse domains ranging from games, to media processing and databases.

D. RQ1: Effectiveness

To answer **RQ1** by demonstrating the effectiveness of WasmDyPa, we conducted an experiment by applying WasmDyPa to the micro-benchmark.

We first executed these benchmarks on WAMR, the Wasm VM directly, and observed the abnormal behaviors of these Wasm programs to trigger either integer overflows or memory bugs. Then, we applied WasmDyPa to these benchmarks, attached analysis hooks on susceptive sites, then executed the instrumented Wasm again to carry out dynamic program analysis.

We presented, in the last column (**WasmDyPa**) of TABLE II, the experimental results. In a summary, WasmDyPa reported 17 potential bugs, among which 2 were false positives, and deemed 3 programs to be bug-free with 1 false negative. Hence, WasmDyPa achieves a 88.24% (15/17) precision and a 93.75% (15/16) recall on WasmBench.

We further investigated root causes leading to false positives and identified one key reason: the type conversion between signed and unsigned integers can confuse WasmDyPa. Specifically, while Wasm arithmetic instructions allow signed

TABLE II: The micro-benchmark, and its experimental results.

Test Case	Vulnerability Type	LoC BI[1]	LoC AI[2]	Instrumentation(s) / per line (ms)	EXE[3]BI Time (s) / per line (ms)	EXE AI Time (s) / per line (ms)	Analysis Time (s) / per line (ms)	WasmDyPa
1	IO2BO_i32add	49	66	0.042 / 0.857	0.041 / 0.837	0.047 / 0.712	0.006 / 0.353	✓
2	IO2BO_i32sub	49	66	0.044 / 0.898	0.042 / 0.857	0.047 / 0.712	0.005 / 0.294	✓
3	IO2BO_i32mul	49	66	0.043 / 0.878	0.041 / 0.837	0.049 / 0.742	0.008 / 0.471	✓
4	IO2BO_i32shl	49	66	0.044 / 0.898	0.042 / 0.857	0.047 / 0.712	0.005 / 0.294	✓
5	IO2BO_i64add	50	82	0.044 / 0.880	0.043 / 0.860	0.049 / 0.598	0.006 / 0.188	✓
6	IO2BO_i64sub	50	82	0.044 / 0.880	0.042 / 0.840	0.047 / 0.573	0.005 / 0.156	✓
7	IO2BO_i64mul	50	82	0.043 / 0.860	0.044 / 0.880	0.048 / 0.585	0.004 / 0.125	✓
8	IO2BO_i64shl	50	82	0.042 / 0.840	0.044 / 0.880	0.046 / 0.561	0.002 / 0.063	✓
9	DF	4,312	16,935	0.061 / 0.014	0.041 / 0.010	0.106 / 0.006	0.065 / 0.005	✓
10	UAF	4,343	17,009	0.052 / 0.012	0.044 / 0.010	0.112 / 0.007	0.068 / 0.005	✓
11	BOF	4,377	17,089	0.052 / 0.012	0.045 / 0.010	0.110 / 0.006	0.065 / 0.005	✓
12	Func_BOF	3,444	13,363	0.048 / 0.014	0.048 / 0.014	0.140 / 0.010	0.092 / 0.009	✓
13	Func_DF	4,259	16,317	0.047 / 0.011	0.044 / 0.010	0.128 / 0.008	0.084 / 0.007	✓
14	Func_UAF	4,252	16,331	0.047 / 0.011	0.041 / 0.010	0.133 / 0.008	0.092 / 0.008	✓
15	Stack-based BOF_1	94	310	0.045 / 0.479	0.050 / 0.532	0.054 / 0.174	0.004 / 0.019	✓
16	Stack-based BOF_2	94	310	0.043 / 0.457	0.040 / 0.426	0.051 / 0.165	0.011 / 0.051	✗
17	Normal $program_1$	7,917	26,528	0.046 / 0.006	0.043 / 0.005	0.139 / 0.005	0.096 / 0.005	✓
18	Normal $program_2$	11,256	39,575	0.053 / 0.005	0.043 / 0.004	0.178 / 0.005	0.135 / 0.005	✓
19	Normal $program_3$	49	125	0.043 / 0.878	0.044 / 0.898	0.051 / 0.408	0.007 / 0.092	✗
20	Normal $program_4$	49	125	0.044 / 0.898	0.043 / 0.878	0.050 / 0.402	0.007 / 0.092	✗

[1] "BI" means before hook instrumentation.
[2] "AI" means after hook instrumentation.
[3] "EXE" means the execution of Wasm program.

operands, its result can be regarded as an unsigned value. Hence, WasmDyPa will issue an "overflown" error, as it treats the result as signed. While WasmDyPa might utilize a more fine-grained algorithm to trace type conversions during execution, implicit conversions can make the precise tracing difficult, leading to false positives.

We next investigated root causes leading to the false negative and identified two key reasons: first, it is impossible for any dynamic program analysis like WasmDyPa to obtain the size of a statically allocated variable (*e.g.*, a buffer `buf[N]` where `N` is a compile-time constant), as such a variable is allocated on stack and its size is implicit in binaries. Hence, WasmDyPa has to conservatively identify a potential buffer overflow with respect to the stack top. Second, WasmDyPa utilized heuristic algorithms to track memory change across library calls. Hence, the conservativity caused by untracked library functions will lead to false negatives. While adding this library will suppress this false negative, a full coverage of all potential library functions in-the-wild is laborious, requiring considerable engineering efforts.

Summary: WasmDyPa achieved a 88.24% precision and a 93.75% recall on the microbenchmark WasmBench, respectively, demonstrating its effectiveness.

E. RQ2: Performance and overhead

To answer **RQ2** by investigating the performance of and overhead introduced by WasmDyPa, we applied WasmDyPa to WasmBench and each Wasm program was executed 10 rounds to calculate the average time.

We present, in TABLE II (the 5th and 8th columns), the performance of WasmDyPa, including: 1) time for static instrumentation on Wasm (**Instrumentation**); and 2) time for dynamic analysis (**Analysis Time**). We give the total execution time as well as the time for executing each line. Experimental results demonstrated that WasmDyPa is efficient in detecting vulnerabilities in Wasm applications: the time spent on instrumentation is approximately 0.05 seconds for each program (or 0.005 to 0.898 milliseconds per line), whereas the analysis time varies from 0.002 to 0.135 seconds for each program (or 0.005 to 0.471 milliseconds per line).

To investigate the overhead WasmDyPa introduced to the Wasm program being analyzed, we measure the execution time of each Wasm program before and after the instrumentation, and present the results in TABLE II (columns **EXE BI Time** and **EXE AI Time**). The results demonstrated that the overhead introduced by WasmDyPa is acceptable: although execution time after instrumentation is increased, the average execution time per line remains the same. We have observed

TABLE III: Experimental results on real-world benchmarks.

Dataset	All	IO[1]	UAF	BOF	DF	SOF[2]	Total
Real-world	941	2	7	5	2	40	56

[1] "IO" means integer overflow.
[2] "SOF" means stack buffer overflow.

that for some test programs (*e.g.*, test cases 15 and 16), the average execution time per line improves. We speculated the reason for such improvement can be attributed to the JIT mode of the WARM VM we used for larger Wasm program.

Summary: WASMDYPA is efficient in detecting vulnerabilities in Wasm programs with acceptable overhead.

F. RQ3: Usefulness

To answer **RQ3** by demonstrating the usefulness of WASMDYPA, we applied it to our second benchmark, 941 real-world Wasm applications from diverse fields such as games, media processing, and database.

We present, in TABLE III, the experimental results on this benchmark. Among the 941 Wasm test cases, WASMDYPA successfully detected 2 integer overflows, and 54 memory-related bugs, which consisted of 7 use-after-free bugs, 5 buffer overflow bugs, 2 double-free bugs, and 40 stack buffer overflow bugs.

Summary: From 941 in-the-wild Wasm programs, WASMDYPA successfully detected 56 bugs, including 2 integer overflow bugs and 54 memory bugs, demonstrating its usefulness on real-world Wasm programs.

G. Case Study

To show WASMDYPA's capability of bug detection in practice and to understand WASMDYPA's effectiveness, we present, as showcases, two bugs detected by WASMDYPA, which belong to IO2BO and heap-based buffer overflows, respectively. We have included, in our open source, a complete list of bugs WASMDYPA successfully identified.

IO2BO bug. We present, in Fig. 6, an integer overflow bug which further leads to an out-of-bound buffer access as documented by CWE-190 [52]. (The initial C code, Wasm code, and corresponding hooks are all presented here.)

This code snippet first allocates a table of size `num_imgs`, which may cause an integer overflow in the multiplication `sizeof(img_t)*num_imgs`, leading to a smaller list being allocated than expected. As a result, subsequent code may trigger out-of-bound bugs, due to the wrong buffer size.

To detect such bugs, WASMDYPA first instruments hooks into the target Wasm program. For example, WASMDYPA places a hook `call $hook_i32_mul` after the original `i32.mul` instruction. After instrumenting hooks, the Wasm program starts execution with the tainted inputs $10 * 2^{12}$ and 2^{20} (we omitted the generation of these inputs here for clarity), and the control flow transfers to the dynamic analysis algorithm `HOOKS_ARITHI32(i32_mul)`, which

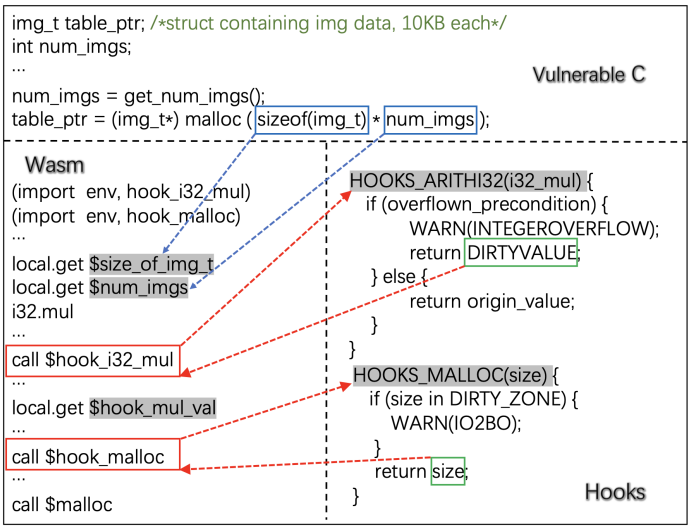


Fig. 6: An integer overflow (leading to buffer overflow) bug detected by WASMDYPA.

will detect this overflow to issue a warning before returning a dirty value `0xC0000000`. Next, the control transfers back to the Wasm program with a dirty return value assigned to `hook_mul_val`, which is further propagated to the memory allocation. Finally, the analysis algorithm `HOOKS_MALLOC`, as the target of the hook `hook_malloc`, will detect the memory overflow by determining the value `size` is in the `DIRTY_ZONE`, thus identifying an IO2BO bug.

Heap-based buffer overflow bug. We present, in Fig. 7, a heap-based buffer overflow bug detected by WASMDYPA, as documented by CWE-122 [69]. The buffer `buf` may be overflowed, as the function `strcpy()` does not perform range checking.

To detect this bug, with the aid of the hook `hook_malloc_info`, WASMDYPA first records, via the analysis algorithm `HOOKS_MALLOC_INFO`, the size and base address of the buffer `buf`. Next, the hook `hook_strcpy` for the function `strcpy` transfers control to the dynamic analysis algorithm `HOOKS_STRCPY`, which determines whether the target memory operation exceeds the buffer range by checking allocated sizes. If that is the case, WASMDYPA would issue a message of buffer overflow with auxiliary information to developers for further investigation and bug fixing.

VII. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work. It should be noted that this work represents the first step toward designing and implementing an effective framework for detecting Wasm bugs via dynamic program analysis.

Wasm instruction support. WASMDYPA supports all Wasm instructions in its version 1.0 specification. However, Wasm is a rapidly growing language with new instructions introduced (*e.g.*, SIMD instructions [70], GC [71], and reference-typed

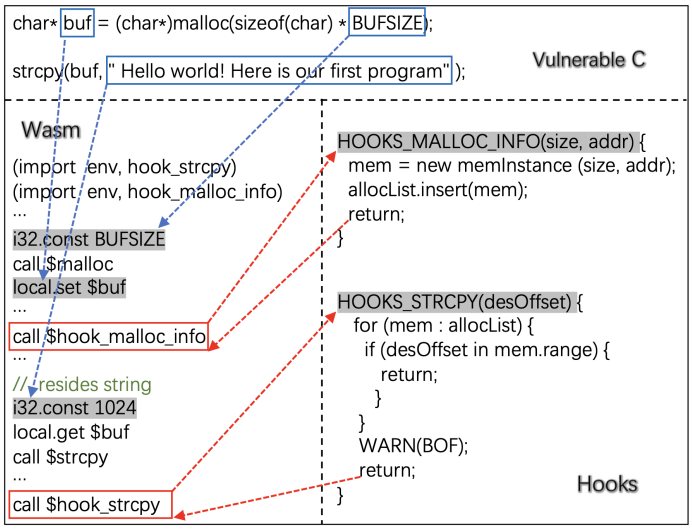


Fig. 7: A buffer overflow bug detected by WASMDYPA.

string proposal [72]). As a result, while we have not encountered such instructions in our experiments, supporting these new instructions of Wasm is essential for WASMDYPA. In the future, we will continue to maintain and extend WASMDYPA, to support newly introduced instructions of Wasm.

Source languages and toolchains. While WASMDYPA's design targets Wasm binary programs and thus is neutral to any specific source languages and compiler toolchains, the current experiments focus on Wasm programs compiled from C/C++ programs by the Clang [73] compiler. Meanwhile, Wasm's support for Rust [74] [75], an emerging safe system programming language, is becoming mature. Hence, it is interesting to investigate how Rust bugs [76] [77] [78] manifest in Wasm programs and how to detect them. We leave this one of our future work.

Wasm VMs. Wasm VM is indispensable for dynamic program analysis. While experimental results showed that our selection of WAMR VM [65] is effective, there are other possible VMs (*e.g.*, Wasmtime [79], Wasmer [80], or WasmEdge [81]). Despite the VM implementation differences, we speculate the selection of VMs only affects the performance and overhead, as both the hooks and analysis algorithms are neutral to VMs.

Other vulnerabilities. While integer overflows and memory vulnerabilities can be detected by WASMDYPA effectively, there are other types of vulnerabilities. Specifically, it is important to investigate concurrency bugs in Wasm programs. To this end, we may start by leveraging recent research progress [82]. We leave it an important future work.

VIII. RELATED WORK

In recent years, there are a significant amount of studies on Wasm security and dynamic program analysis. However, the work in this paper stands for a novel contribution to these fields.

Empirical security studies. Several empirical studies have been conducted on Wasm. Lehmann et al. [11] conducted an empirical study on Wasm binary security. Musch et al. [83] characterized the prevalence of Wasm in the wild. Wang et al. [22] conducted an empirical study on Wasm runtimes to analyze bug root causes. Romano et al. [21] presented an empirical study on Wasm compilers to analyze bug root causes and fixing strategies. However, a key difference between these studies and our work in this paper is that we focus on the detection of the vulnerability in Wasm. Hence, our work is orthogonal to existing studies and supplements them.

Program analysis. Program analysis on Wasm has been extensively studied. Haas et al. [3] proposed a small-step operational semantic and a type system to check the safety of stack-based operations for Wasm. Stiévenart et al. [84] proposed an information flow analysis algorithm for Wasm programs and Lopes et al. [13] proposed Wasmati, a vulnerability detection framework for Wasm, based on the code property graph. Chen et al. [6] proposed a fuzzing framework Wasai for Wasm smart contracts. Szanto et al. [85] and Fu et al. [86] performed taint analysis on Wasm to track the propagation of data and detect possible input vulnerabilities. A key limitation of existing studies is that they have not performed comprehensive study on Wasm integer overflows and memory vulnerabilities. On the contrary, this work, for the first time, investigates and detects these vulnerabilities and their mitigation.

Dynamic analysis. Dynamic analysis has been extensively studied. Agrawal et al. [87] proposed dynamic slicing as a complement to static slicing [88] with efficient debugging and testing capability. Newsome et al. [49] proposed dynamic taint analysis featuring fast automatic attack detection and filtering mechanisms for x86 binaries. Bond et al. [89] proposed an efficient origin tracking of unusable values. While these works focus on various dynamic analyses, they cannot apply to Wasm due to the language feature discrepancies. On the contrary, our work can detect vulnerabilities in real-world Wasm programs.

Dynamic analysis of Wasm. Many dynamic analyses for WebAssembly have been proposed, including a taint analysis [90], a cryptomining detector [91] and a dynamic analysis framework [14]. Some of these analyses have been confined to browsers (*e.g.*, by modifying the V8 engine [92]). However, a major limitation of existing Wasm dynamic analysis studies and tools is that they can only be conducted on browsers. On the contrary, in this work, we extend the dynamic analysis out of the browsers by deploying it on a Wasm standalone VM, achieving both efficiency and portability.

IX. CONCLUSION

This paper presented WASMDYPA, the first infrastructure to detect Wasm bugs by dynamic program analysis. WASMDYPA consists of tainted input generation, hook instrumentation, and dynamic analysis algorithm as security plugins. The prototype implementation and evaluations demonstrated that WASMDYPA is effective in detecting real-world Wasm bugs with acceptable performance and overhead. Overall, the work in this paper represents a first step toward dynamic Wasm vulnerability detection, making Wasm not only an efficient but also a safer programming language.

REFERENCES

[1] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening webassembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[2] "Webassembly," https://webassembly.org/.

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

[4] B. McFadden, T. Lukasiewicz, J. Dileo, and Engler, "Security chasms of wasm," Tech. Rep., Aug. 2018.

[5] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, Jan. 2019.

[6] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: Uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual South Korea: ACM, Jul. 2022, pp. 703–715.

[7] "Apache/apisix: The cloud-native api gateway," https://github.com/apache/apisix.

[8] "Introduction - embedded webassembly (wasm)," https://embedded-wasm.github.io/book/.

[9] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *Web Engineering*, ser. Lecture Notes in Computer Science, M. Brambilla, R. Chbeir, F. Frasincar, and I. Manolescu, Eds. Cham: Springer International Publishing, 2021, pp. 328–336.

[10] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *USENIX Security*, 2020, p. 19.

[11] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*. Ljubljana Slovenia: ACM, Apr. 2021, pp. 2696–2708.

[12] "Cwe - cwe-680: Integer overflow to buffer overflow (4.11)," https://cwe.mitre.org/data/definitions/680.html.

[13] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, Jul. 2022.

[14] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 1045–1058.

[15] "Wabt," https://webassembly.github.io/wabt/doc/wasm2wat.1.html.

[16] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.

[17] "Webassembly design," WebAssembly, May 2023.

[18] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses."

[19] "Cwe - cwe-415: Double free (4.11)," https://cwe.mitre.org/data/definitions/415.html.

[20] "Cwe - cwe-416: Use after free (4.11)," https://cwe.mitre.org/data/definitions/416.html.

[21] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia: IEEE, Nov. 2021, pp. 42–54.

[22] Y. Wang, "A comprehensive study of webassembly runtime bugs."

[23] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," Oct. 2021.

[24] "Types — webassembly 2.0 (draft 2023-04-24)," https://webassembly.github.io/spec/core/syntax/types.html.

[25] "Security - webassembly," https://webassembly.org/docs/security/.

[26] "Going public launch bug · issue #150 · webassembly/design," https://github.com/WebAssembly/design/issues/150.

[27] "Roadmap - webassembly," https://webassembly.org/roadmap/.

[28] "Webassembly core specification," https://www.w3.org/TR/wasm-core-1/.

[29] "World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation," https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en.

[30] "Standardizing wasi: A system interface to run webassembly outside the web – mozilla hacks - the web developer blog," https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.

[31] "Webassembly high-level goals - webassembly," https://webassembly.org/docs/high-level-goals/.

[32] M. Kim, H. Jang, and Y. Shin, "Avengers, assemble! survey of webassembly security solutions," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. Barcelona, Spain: IEEE, Jul. 2022, pp. 543–553.

[33] "Homepage — wasmcloud," https://wasmcloud.com/.

[34] R. Liu, L. Garcia, and M. Srivastava, "Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, 2021, pp. 94–105.

[35] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-based deep behavioral detection of web cryptomining with coinspy," in *Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web*. San Diego, CA: Internet Society, 2020.

[36] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "Eosafe: Security analysis of eosio smart contracts," p. 19.

[37] W. Bian, W. Meng, and Y. Wang, "Poster: Detecting webassembly-based cryptocurrency mining," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. London United Kingdom: ACM, Nov. 2019, pp. 2685–2687.

[38] "Serverless edge compute solutions — fastly," https://www.fastly.com/products/edge-compute.

[39] "Scalar.video - let your creativity run wild on an infinite canvas." https://www.url.ie/a.

[40] "Torch2424/wasmboy: Game boy / game boy color emulator library, written for webassembly using assemblyscript. demos built with preact and svelte." https://github.com/torch2424/wasmBoy.

[41] T. Ball, "The concept of dynamic analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 216–234, Oct. 1999.

[42] M. D. Ernst, "Static and dynamic analysis: Synergy and duality."

[43] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: Association for Computing Machinery, Mar. 2010, pp. 167–178.

[44] J. Park, B. Choi, and S. Jang, "Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments," *International Journal of Parallel Programming*, vol. 48, no. 6, pp. 1032–1060, Dec. 2020.

[45] T. M. Chilimbi and V. Ganapathy, "Heapmd: Identifying heap-based bugs using anomaly detection," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: Association for Computing Machinery, Oct. 2006, pp. 219–228.

[46] X. Yu, S. Han, D. Zhang, and T. Xie, "Comprehending performance from real-world execution traces: A device-driver case," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 193–206, Feb. 2014.

[47] J. Fiedor, M. Mužikovská, A. Smrčka, O. Vašíček, and T. Vojnar, "Advances in the anaconda framework for dynamic analysis and testing of concurrent c/c++ programs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 356–359.

[48] "A survey of dynamic analysis and test generation for javascript — acm computing surveys," https://dl.acm.org/doi/10.1145/3106739.

[49] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software."

[50] C. Watt, "Mechanising and verifying the webassembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Los Angeles CA USA: ACM, Jan. 2018, pp. 53–65.

[51] "Emscripten-core/emscripten," emscripten-core, May 2023.

[52] "Cwe - cwe-190: Integer overflow or wraparound (4.11)," https://cwe.mitre.org/data/definitions/190.html.
[53] "Openssh," https://www.openssh.com/.
[54] "Wasm-tools/crates/wasm-mutate at main · bytecodealliance/wasm-tools," https://github.com/bytecodealliance/wasm-tools.
[55] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 95–105.
[56] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 4:1–4:36, Feb. 2020.
[57] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 261–278, Feb. 2021.
[58] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 65–76, Jun. 2015.
[59] "Warf - webassembly runtimes fuzzing project," FuzzingLabs, May 2023.
[60] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in c/c++," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 1, pp. 2:1–2:29, Dec. 2015.
[61] K. Haßler and D. Maier, "Wafl: Binary-only webassembly fuzzing with fast snapshots," in *Reversing and Offensive-Oriented Trends Symposium*. Vienna Austria: ACM, Nov. 2021, pp. 23–30.
[62] H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. Singapore Republic of Singapore: ACM, Apr. 2015, pp. 483–494.
[63] S. Huang, J. Guo, S. Li, X. Li, Y. Qi, K. Chow, and J. Huang, "Safecheck: Safety enhancement of java unsafe api," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 889–899.
[64] "American fuzzy lop," Google, May 2023.
[65] "Bytecodealliance/wasm-micro-runtime: Webassembly micro runtime (wamr)," https://github.com/bytecodealliance/wasm-micro-runtime.
[66] B. Ray, D. Posnett, P. Devanbu, and V. Filkov, "A large-scale study of programming languages and code quality in GitHub," *Communications of the ACM*, vol. 60, no. 10, pp. 91–100, Sep. 2017.
[67] "Webassembly/wabt: The webassembly binary toolkit," https://github.com/WebAssembly/wabt/tree/main.
[68] "Cwe - cwe-658: Weaknesses in software written in c (4.11)," https://cwe.mitre.org/data/definitions/658.html.
[69] "Cwe - cwe-122: Heap-based buffer overflow (4.11)," https://cwe.mitre.org/data/definitions/122.html.
[70] "Simd proposal for webassembly," WebAssembly, Jun. 2023.
[71] "Gc proposal for webassembly," WebAssembly, Jun. 2023.
[72] "Stringref/overview.md at main · webassembly/stringref," https://github.com/WebAssembly/stringref.
[73] "Llvm-mirror/clang: Mirror kept for legacy. moved to https://github.com/llvm/llvm-project," https://github.com/llvm-mirror/clang.
[74] "Rust programming language," https://www.rust-lang.org/.
[75] "Webassembly," https://www.rust-lang.org/what/wasm.
[76] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," Feb. 2021.
[77] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. London UK: ACM, Jun. 2020, pp. 763–779.
[78] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 246–257.
[79] "Bytecodealliance/wasmtime: A fast and secure runtime for webassembly," https://github.com/bytecodealliance/wasmtime.
[80] "Wasmerio/wasmer: the leading webassembly runtime supporting wasi and emscripten," https://github.com/wasmerio/wasmer.
[81] "Wasmedge/wasmedge: Wasmedge is a lightweight, high-performance, and extensible webassembly runtime for cloud native, edge, and decentralized applications. it powers serverless apps, embedded functions, microservices, smart contracts, and iot devices." https://github.com/WasmEdge/WasmEdge.
[82] C. Watt, A. Rossberg, and J. Pichon-Pharabod, "Weakening webassembly," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, Oct. 2019.
[83] R. Perdisci, M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New kid on the web: A study on the prevalence of webassembly in the wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, vol. 11543, pp. 23–42.
[84] Q. Stievenart and C. D. Roover, "Compositional information flow analysis for webassembly programs," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 13–24.
[85] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," Jul. 2018.
[86] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," Feb. 2018.
[87] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990.
[88] M. Harman and R. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.
[89] M. D. Bond, N. Nethercote, and S. W. Kent, "Tracking bad apples: Reporting the origin of null and undefined value errors."
[90] Q. Stiévenart, "Wassail," Apr. 2023.
[91] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. Vigna, "Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Toronto Canada: ACM, Oct. 2018, pp. 1714–1730.
[92] "V8 javascript engine," https://v8.dev/.