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 safe and platform-independent binary set that has been deployed in many security critical areas 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 unpredictable security hazards to WebAssembly applications.**

In this paper, we propose the first automated bug detection framework for WebAssembly programs based on dynamic program analysis on WebAssembly binaries directly. To realize the whole process, we present WASMDYPA, the first dynamic bug detection system, consisting of 3 primary components: 1) a fuzzing tool to generate taint inputs for WebAssembly binaries; 2) a static instrumentation module providing extensible interfaces to collect and record runtime information; and 3) program analysis plugins to dynamically detect security vulnerabilities. We have implemented a software prototype for WASMDYPA, and have conducted experiments to evaluate the effectiveness, usefulness and performance of our approach. 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 binaries, including 2 integer overflows and 54 memory bugs.

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

I. INTRODUCTION

Today's cloud or edge computing infrastructures put forward higher requirements for the safe execution of the 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, which 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 towards defining a secure binary distribution format, existing studies [10] 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 [11]. Despite this urgent needs, existing studies and tools [12] [13] [14] for Wasm cannot detect these vulnerabilities caused by type system design defects.

Second, to guarantee control flow integrity [15], Wasm introduced a fine-grained memory model [16] 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) [17] based attacks effectively, existing studies [10], unfortunately, have revealed Wasm programs are still vulnerable and exploitable. Worse yet, Wasm does not support garbage collections but rely on manual memory management, thus might lead to the notorious memory vulnerabilities such as double free (DF) or use-after-free (UAF). Despite of the fact that current studies have empirically studied Wasm compilers [18] or runtimes bugs [19], as well as mitigations (*e.g.*, stack canary [20]), a systematic study of effective bug detection for Wasm is still missing.

Recognizing this security criticality and urgency, we propose using dynamic analysis approach to detect potential vulnerabilities by leveraging runtime information. Specifically, we argue that making a dynamic analysis framework dedicated for 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 [21]) or dynamic (*e.g.*, sandboxing [22]) 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, by leveraging runtime information which is hard to obtain in static analysis. WASMDYPA consists of three main components: 1) a Wasm compatible taint test case generation component, which generates taint test cases covering sink sites; 2) static instrumentation hooks, which tracks and collects 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 the tainted test generation by utilizing static binary instrumentation techniques, to emulate a tainted input by imposing restriction on the potential sink sites. Furthermore, to improve path coverage, we leveraged and extended existing path-coverage oriented Wasm fuzzers. Second, we implemented the instrumentation of hooks by a dedicated compiler rewriting pass, to traverse the Wasm abstract syntax tree data structures. Moreover, we utilize a validator to check the semantic consistency before and after the instrumentation. Third, we implement diverse 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 have conduct systematic evaluation of WASMDYPA, in terms of its effectiveness, usefulness, and performance, on both micro- and real-world benchmarks. First, we write buggy code on purpose to create the micro-benchmark as a set of ground truth and test WASMDYPA on them to demonstrate the effectiveness. Second, we measure the performance of WASMDYPA on mirco-benchmark in terms of 4 aspects: 1) the time that static instrumentation takes; 2) the execution time of Wasm before instrumentation; 2) the execution time of Wasm after instrumentation; and 3) the analysis time consumption. Third, we extracted 941 real-world Wasm both from wild and CWE to measure the usefulness of WASMDYPA. Our experiments show that WASMDYPA can detect all integer overflow and memory-safety issues in micro-benchmark and is capable of performing bug finding in real-world scenarios, where it detected a total of 56 previously unknown bugs including 2 integer overflow issues, and 54 memory-safety issues.

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 ananlysis.
- **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://github.com/abcdef.

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. Section IV and V presents 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 [23] and became a de-factor standard language in browsers in 2017 [24]. With the first complete formal definition of Wasm released in 2018 [25], Wasm was announced as an official (and the fourth) Web standard by the W3C [26] in 2019. With the design of WebAssembly System Interface (WASI) [27] and the standard version 2.0 draft of Wasm [25], 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 [28]. First, to ensure program safety, Wasm incorporates secure features such as strong typing, sandboxing isolation and secure control flow [3] [22]. 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, makes 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 domain, 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 [29], IoT [30], blockchain [6] [31] [32] [33], edge computing [34], video transcoder [35], and game engines [36]. 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 [37] is a well-established technique to analyze properties of software or systems by observing their runtime behavior. Unlike static analysis [38], which examines the source code of an application without executing it, dynamic

analysis examines how an application behaves in real-world scenarios, thus 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. And the analysis results can be further used to investigate and identify diverse program defects such as concurrency bugs [39], memeory bugs [40], and performance issues [41].

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

III. Motivation and Threat Model

In this section, we first present an overview of Wasm security (§ III-A) and a motivating example (§ 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 [42]: integer overflows, and memory corruptions.

Integer overflows. Although Wasm's strong type system guarantees type safety [3] [43], 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) [11], when the integer is used in in memory allocation. Despite these urgent security needs, existing studies and tools [12] [13] [14] for Wasm cannot detect vulnerabilities caused by integer overflows.

Memory corruptions. Although Wasm memory module isolates managed data and function return address, to prevent return oriented progmmaing-based attack [17], 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 upwards 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` [44]) have been hardened against a variety of metadata corruption attacks, such attacks are also 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 free memory issue we adapted from real-world vulnerabilities CWE-190 [45] and CWE-415 [46], 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 CVE [45] in OpenSSH 3.3 [47]. This code snippet allocates memory for a `char*` array to store network packets. However, if the number of packet `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 [45].

```
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 [46].

While static analysis is difficult to detect 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 is trustworthy in producing semantics equivalent Wasm programs from sources. One the one hand, with considerable efforts in development and testing, compiler toolchains for Wasm are becoming mature [48]. On the other hand, many studies are devoted to testing or fuzzing compilers to detect potential bugs [49] [50]. 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 [19], and fuzzing [51]). On the other hand, Wasm VM security studies are orthogonal to and supplement the the security study of Wasm programs in this work.

We assume Wasm programs may contain bugs, 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++ [52]. 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 studies in this work. Third, WasmDyPa should achieve low overhead in terms of the instrumentation time, execution time, and analysis time.

B. Overall Architechture

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 retrieves and analyzes runtime data to detect potential vulnerabilities and bugs.

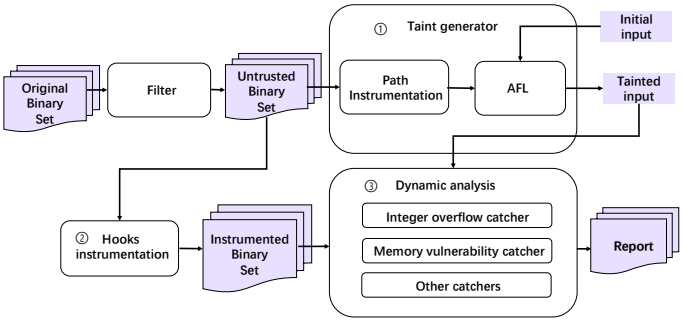


Fig. 3: The architecture of WasmDyPa.

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, with some irrelevant instructions omitted for brevity.

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 $\rho* \rightarrow \rho*$.

A instruction i consists of binary/unary operations, memory load/stores, structured control flows, and function invocation/return. Wasm instructions demonstrated three distinct properties: first, Wasm is a stack-based VM in that operands and result of an operation is 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 strong typed in specifying the expected type in the opcode (*e.g.*, `i32.abs`), which facilitating binary-level type checking. Third, Wasm support structured control flows (*e.g.*, `if` or `loop`), making compilation to Wasm easier.

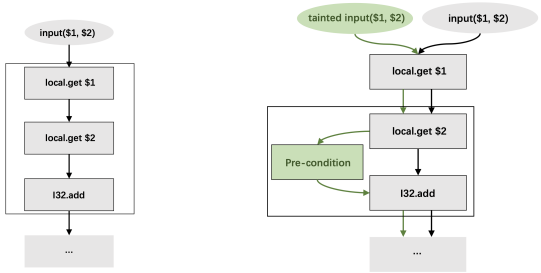
D. Tainted Input Generation

Effective dynamic detection of potential vulnerabilities often requires the availability of tainted input data [53], whose propagation can trigger deep bugs. Hence, to detect Wasm bugs dynamically, a well-designed tainted input generator is essential. However, current Wasm fuzzing tools (*e.g.*, WAFL [54], or Fuzzm [20]) focus on the coverage of different 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 assertions as preconditions.

Pre-condition instrumentation. The pre-condition instrumentation takes as input a Wasm program, 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, 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

Val. Type	ρ	::=	`i32` | `i64` | `f32` | `f64`
Func. Type	σ	::=	$\rho^* \rightarrow \rho^*$
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` *label*
		|	`br_if` *label* | `br_table` *label*$^+$
		|	`select` | `memory.grow`
		|	ρ`.const` c | ...
Function	f	:=	$\sigma\ x\{i^*\}$
Module	m	:=	f^*

Fig. 4: Core syntax of Wasm language.



(a) Control flow before instrumentation

(b) Control flow after instrumentation

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

overflows for the `i32.add` instruction (Fig. 5a). To tackle this problem, we instrument the candidate instruction with pre-condition. For the addition instruction in this example, we instrument the 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 to meet 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.)

E. Hook Instrumentation

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

TABLE I: Overflow predicates used as pre-conditions.

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

We proposed a hook instrumentation strategy in a syntax-directed manner following the instruction i in Fig. 4. For example, for the function invocation instruction `call` f, we place two hooks `before_call` and `after_call` before and after the call instruction, to collect function arguments and return values, respectively. For the purpose of this paper, we present detailed designs of two specific hooks next: integer hooks and memory hooks.

Integer hooks. Binary hooks track the input and output of binary operations. However, to accurately detect integer overflow and identify the sink sites where a poisoned value would be used without excessive resource occupation, we introduced several conceptions of 3 design goals: accuracy, efficiency, and consistency.

First, the primary requirement of the integer overflow checker is accuracy. At runtime, upon the occurrence of an integer overflow caught by our algorithms (§ section IV-F), WASMDYPA does not halt the programs immediately when the overflown values have minor or no influence on programs (e.g., dead overflown value). Instead, we track the data-flow propagation of overflown value and issue an integer overflow warning if the value flows into sinks and causes damage, such as buffer overflow.

Second, to achieve efficiency, a method should be applied to mitigate the time cost of recording data-flow propagation (*e.g.*, a constant factor of time consumption in traditional taint analysis [55] [56] and traditional dynamic information flow tracking [57] [58]). On top of that, we replace the overflown value with a *dirty* value, *i.e.*, a value from a rarely used number field, *dirty-zone*, a conception in prior work [59], to detect IO2BO. It should be noted that a *dirty* value should meet 3 prerequisites. First, replacing overflown values with *dirty* values will not change program semantics and behavior. Second, the *dirty* value should remain in *Dirty-zone* after arithmetic operations, *i.e.*, keeping its dirty property which can be further caught by our algorithm (§ section IV-F). Third, the range of *dirty-zone* should depends on different integer overflows (*e.g.*, IO2BO, infinite loop). To clearly demonstrate how we meet the derequisitions, we take IO2BO as an example. First, dirty values should be derived from a range of very large and rarely used integer values to retain semantic consistency considering the statistical pattern of the size of memory allocation behavior. As the previous study rigorously proves that the designated initial dirty value (*i.e.* 0xC0000000) can meet acquirement among 32-bit arithmetic

operations, and we select our *dirty-zone* as [0xA0000000, 0xE0000000), since it is 256 times larger than the regular interval of allocation size. It is notable that we only considered 32-bit arithmetic because the type of memory address is i32, and there must be a truncation from i64 to i32 to satisfy the memory allocation *i.e.*, memory.grow, and we assumed that such an abnormal allocation behavior would rarely happen in our study based on manual filtering. Second, while utilizing the *dirty-value* derived from *dirty-zone*, we have solid data support demonstrates that even under a sequence of 10 arithmetic operations, the value remains dirty with the probability around 0.9999776 according to the previous study.

Third, to achieve semantic consistency, we have to instrument Wasm instructions around sites to preserve the context. The basic concept is storing the input and output of an arithmetic instruction in freshly generated local variables without changing the operation stack status by leveraging `local.tee`, which stores the value at the top of the stack into the local variable and keeps the value remain at the top of the stack at the same time. Then, push them into the operation stack again when invoking the binary hooks to play the role of parameters. It is notable that we have to drop the original result of arithmetic and replace them with our *dirty* value in further propagation.

Memory hooks. In general, we designed call hooks to track the input and output of direct call operations, and memory access hooks to trace the memory address, both with the validated instrumentation to preserve context. Similar to the binary hooks, to accurately detect memory vulnerabilities, we still obey the 3 principles: accuracy, efficiency, and consistency.

To achieve accuracy while retaining efficiency, hooks should be selectively attached to memory-related function call. As we mentioned before, memory vulnerabilities can pass through high-level language and mainstream of the resource is mainly written in C/C++, hence, we need to track the function with name such as `malloc`, `free`, and `strcpy` which are widely used and unsafe in C/C++. However, we need to attach the memory access hooks to every load and store instruction which is a trade-off between accuracy and efficiency. It is notable that we have to obtain the size and stack pointer of the unmanaged stack in linear memory, and we assume the stack pointer is marked as a global variable with index zero based on our observation of our benchmark, and that is, hooks attached to global instructions, *i.e.*, `global.get` and `global.set` is a must. To achieve consistency, we applied the same strategy depicted in binary hooks to preserve context.

F. Dynamic Analysis Algorithm as Security Plugins

We presented the heuristic dynamic analysis algorithms in terms of integer overflow plugin and memory-safety plugin.

Integer overflow. The Algorithm 1 demonstrates the principle of integer overflow analysis, and the key idea for it is to: 1) identify the overflown value; and 2) replace the overflown value with *dirty* value.

To realize this key idea, `overflowCacther` takes in a set of instructions $\mathbb{I}_{wat}$ as well as the operation stack $Stack$,

Algorithm 1: Integer overflow detection algorithm.

Input: $\mathbb{I}_{wat}$: the instruction in a Wasm module, $Stack$: the operation stack
Output: $Stack_u$: the update of stack

```
1  Function overflowCatcher(I_wat, Stack):
2      T ← getInstructionType (I_wat);
3      if T == arith then
4          L_info ← getRuntimeInfo (Stack);
5          isOverflown ← checkOverflown (L_info);
6          if isOverflown then
7              warning(isOverflown);
8              Stack_u ← stackUpdate(Stack,dirtyValue);
9              return Stack_u;
10         else
11             Stack_u ← stackResume (Stack, OriginValue);
12             return Stack_u;
13     else
14         Stack_u ← stackResume (Stack, L_info);
15         return Stack_u;
```

Algorithm 2: IO2BO detection algorithm.

Input: $\mathbb{I}_{wat}$: the instruction in a Wasm module, $Stack$: the operation stack
Output: $Stack_u$: the update of stack

```
1  Function IO2BOCatcher(I_wat, Stack):
2      T ← getInstructionType (I_wat);
3      if T == memory_alloc then
4          L_info ← getRuntimeInfo (Stack);
5          isInDirtyZone ← checkStillDirty (L_info);
6          if isInDirtyZone then
7              abort(IO2BO);
8          else
9              Stack_u ← stackResume (Stack, L_info);
10             return Stack_u;
11     else
12         Stack_u ← stackResume (Stack, L_info);
13         return Stack_u;
```

and returns the updated operation stack $Stack_u$. This function consists of 3 key steps: first, the overflow catcher traverses each instruction and identifies the arithmetic operation (*i.e.*, shown in the first column of Table I), as well as obtains their inputs and outputs from $Stack$ (line 2 to 4). Second, for each arithmetic operation, the catcher analyzes it's result and inputs by applying the overflow condition (shown in second column of Table I) and emits informative diagnosis if an overflown happens (line 5 to 7). Third, the catcher replaces the overflown value with a *dirty* value and place it on the stack top (line 8 to 9). It is notable that we have to keep the consistency of the

stack's status which has been discussed before (§ IV-E).

The detailed IO2BO detection process is given in Algorithm 2 with the same meaning of inputs and outputs depict in Algorithm 1. It is notable that when catching the pre-allocated memory size of a memory allocation instruction from $Stack$ (line 2 to 4), the catcher further distinguishes the *dirty* address by checking whether it is in *dirty-zone* (line 5) and aborts the program with an informative diagnosis when IO2BO happens (line 7).

Memory vulnerability. The Algorithm 3 demonstrates the algorithm of memory vulnerability analysis, and the key idea for it is to: 1) records every memory address and; 2) validates each memory access.

To realize this idea, `MemVulnerCatcher` takes in the $HeapBase$ to record the address heap base, lists $SPList$ and $SPSize$ to record stack frame and stack size of each Wasm function, as well as lists $HPList$ and $HPSize$ to record available heap memory, respectively, others are same as Algorithm 1. This instruction consists of 4 steps: first, the catcher obtains the unique heap base as a boundary between the stack segment and heap segment (line 8). Second, the catcher records every stack allocation (line 6 to 18) and heap allocation information (line 21 to 23). Third, the catcher validates every memory access and outputs diagnosis in terms of heap-based and stack-based memory vulnerability (line 24 to 30). Fourth, it retrieves the formal stack frame and size after existing current function (line 31 to 35).

Algorithm 3: Memory-safety detection algorithm.

Input: $HeapBase$: the address of heap base, $SPList$: a list records the address of stack pointer, $STSize$: a list records the size of stack, $SPList$: a list records the address of heap, $STSize$: a list records the size of heap piece; others are asme as Algorithm 1

```
1  Function MemVulnerCatcher(I_wat, Stack, HeapBase, SPList, SPSize):
2      T ← getInstructionType (I_wat);
3      if T == globalGetOp then
4          Index ← getGlobalIndex (I_wat);
5          isSP ← isStackPointer(Index);
6          if isSP then
7              SP_current = getCurrentStackPointer();
8              HeapBase ← isSet() ? HeapBase :
                 setHeapBase(HeapBase, SP_current);
9          else
10             return;
11     else if T == subOp then
12         Val ← getFirstOperands(I_wat);
13         eqSP ← eqStackPointer(Val,SP_current);
14         if eqSP then
15             ST_size ← getSecondOperands(I_wat);
16             SP_base ← getResult(I_wat);
17             pushSPList(SP_base, SPList);
18             pushSPSize(ST_size, STSize);
19         else
20             return;
21     else if T == heap_alloc then
22         pushHPList(getMemBase(I_wat));
23         pushHPSize(getMemSize(I_wat));
24     else if T == memAccess then
25         Addr ← getAddress(Stack);
26         isLegal ← validateAddress(SP_base, ST_size,
             HeapBase, HPList, HPSize, Addr);
27         if isLegal then
28             continue;
29         else
30             warning(MemVulnerability);
31         else if T == return then
32             popSPList();
33             popSTSize();
34             SP_base ← getSPListEnd();
35             ST_size ← getSTSizeEnd();
36     else
37         return;
```

V. WasmDyPa Implementation

To validate the system design, we have implemented a software prototype for WasmDyPa, consisting of three main components: 1) a fuzzing tool to generate tainted data to trigger potential vulnerabilities; 2) a hooks instrumentation to catch essential runtime information; and 3) a dynamic checker to catch and detect the malicious exploitation of resources. For fuzzing, we leveraged Fuzzm [20] , a fuzzing tool for Wasm input generation adopted from AFL [60], to generate tainted input for benchmarks. For program analysis, we utilized the hooks provided by Wasabi to catch the runtime information, but to realize our research goals, we modified the form of hooks and perform extra instrumentation that validated by the Wasm static type check. Furthermore, in order to provide a high-scalability interface while maintaining efficiency, we ported Wasabi from browsers to a popular used Wasm virtual machine, *i.e.*, wasm-micro-runtime (WAMR) [61] with 3.8k stars, which is written in C/C++, and implemented our runtime algorithms as a C library through following the native function interface standard provided by WAMR, to support our design goals. Last but not least, we leverage `wasm-validate` from the WABT [62] which offers some well-formedness guarantees and checks that the code is type correct to validate the consistency of instrumented Wasm code.

In conclusion, we ported 3 state-of-the-art analysis tools and a standalone Wasm virtual machine: 1) Wasabi, a dynamic runtime tracker of Wasm; 2) Fuzzm, a fuzzing tool designed for Wasm; 3) WABT, a Wasm toolkit support validation of a file in the WebAssembly binary format; adn 4) WAMR, a high profile Wasm virtual machine.

VI. Evaluation

In this section, we present experiments to evaluate WasmDyPa. We first present the research questions guiding the

experiments (Section VI-A), then discuss the benchmark we adopted (§ VI-C), and finally discuss the results in terms of the effectiveness, usefulness and performance of WASMDYPA (§ VI-D to § VI-F). Finally, we present the case study (§ 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 real-world bugs in Wasm programs?

RQ2: Usefulness. As WASMDYPA is proposed to secure practical Wasm applications, is it useful to large and real-world applications?

RQ3: Performance. As WASMDYPA makes of static instrumentation, and dynamic analysis to detect vulnerabilities, what is the performance of it?

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 programs, consists of 61 weaknesses in software written in C collected from CWE and 880 Wasm test cases.

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 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 bench for integer overflow and memory vulnerabilities program analysis. As shown in Table II, We manually constructed a microbenchmark consisting of 18 test cases with diverse types of vulnerabilities (as presented in the second column of Table II), including IO2BO, double-free, buffer overflow, *etc*.

Currently, WasmBench only includes integer overflow and memory vulnerabilities. We will maintain and augment it by including more benchmarks and test cases while covering other vulnerabilities. The current version of WasmBench is included in our open-source package for WASMDYPA.

Real-world Wasm programs. Our real-world Wasm dataset consists of 941 unique Wasm binaries in wild, by referring the dataset in prior work [42] and 61 weaknesses in software written in C collected from CWE Evaluating WASMDYPA on this benchmark demonstrates the effectiveness of WASMDYPA on diversity real-world Wasm applications ranging from games, over media processing to database.

D. RQ1: Effectiveness

To answer **RQ1** by demonstrating the effectiveness of WASMDYPA, we conducted an experiment on micro-benchmarks in two steps. First, we construct a group of micro-benchmarks consisting of Wasm programs, and manually inject common vulnerabilities into the native code of these applications. For example, we inject poisoned memory allocation, that is, by imposing the result of overflown value to memory allocation instruction *i.e.*, `memory.grow` and memory allocation function *i.e.*, `malloc`, the overflown value that smaller than the expected allocation size can lead to insufficient memory allocation and further result in a buffer overflow.

Second, after injecting these vulnerabilities, we executed these benchmarks on WAMR, and observed the abnormal behavior of Wasm program. Then, we applied WASMDYPA to these benchmarks, attached analysis hooks on susceptive sites, and then executed the instrumented Wasm again to carry out dynamic analysis based on the runtime information collected by hooks. We observed WASMDYPA successfully detected most of these attacks and reported informative information. For example, for IO2BO, the WASMDYPA reports the integer overflow and catch the *dirty* value been propagated at the memory allocation site.

As a result, WASMDYPA achieves 88.24% precision and 93.75% recall on **WasmBench** with 2 false positive and 1 false negative. False positives happen when converting signed values to unsigned values, as integer overflow is detected based on the signed value, which is adopted by most Wasm arithmetic operations. The situations for the false negative can be concluded in two aspects. First, it is impossible for WASMDYPA to obtain the size of a statically allocated memory without modifying the allocator's behavior, hence WASMDYPA conservatively identify a stack-based buffer overflow that exceeds the stack top, *i.e.*, a stack-based buffer overflow can bypass WASMDYPA if it only compromises the data within the stack range it belongs to. Second, WASMDYPA is designed with heuristic algorithms, with the ability to track every memory change. Hence, we trace the representative vulnerable functions such as `strcpy`, to demonstrate the effectiveness. However, a full coverage of all functions in the real world is not possible, leading to the false negative.

Although it may not accurately depict real-world applications, WASMDYPA has incorporated common program analysis features in its design to evaluate the soundness and accuracy of analysis. Therefore, the numbers generated by **WasmBench** still hold significance in assessing the effectiveness of WASMDYPA for Wasm safety analysis.

Summary: WASMDYPA achieved 88.24% precision on our real-world systems, respectively, with a 93.75% recall, hence promising effectiveness.

E. RQ2: Usefulness

To answer **RQ3** by demonstrating the usefulness of WASMDYPA, we applied it to our second benchmark, the real-world Wasm applications from representative fields such as games, media processing, database, and also compiled from vulnerable C/C++. It should be noted that the usefulness of WASMDYPA was verified on these applications by injecting attacks intentionally, such as test cases that simulate tainted data. As shown in Table III, the dataset contains a total of 941

TABLE II: Experimental results on micro-benchmarks.

Test Case	Vulnerability Type	LoC BI	LoC AI	Instrumentation(s) / per line (ms)	EXE 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	0.044 / 0.898	0.042 / 0.857	0.047 / 0.712	0.005 / 0.294	✓
2	IO2BO_i32sub	49	66	0.043 / 0.978	0.045 / 0.918	0.048 / 0.727	0.003 / 0.176	✓
3	IO2BO_i32mul	49	66	0.042 / 0.857	0.047 / 0.959	0.048 / 0.727	0.001 / 0.059	✓
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	0.044 / 0.880	0.046 / 0.561	0.002 / 0.063	✓
6	IO2BO_i64sub	50	82	0.045 / 0.900	0.043 / 0.860	0.044 / 0.537	0.001 / 0.031	✓
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	0.045 / 0.900	0.043 / 0.860	0.047 / 0.573	0.004 / 0.125	✓
9	Inter_BOF	3,444	13,363	0.048 / 0.014	0.044 / 0.013	0.064 / 0.005	0.020 / 0.002	✓
10	DF	4,312	16,935	0.052 / 0.012	0.043 / 0.010	0.055 / 0.003	0.012 / 0.001	✓
11	UAF	4,343	17,009	0.049 / 0.011	0.043 / 0.010	0.059 / 0.009	0.016 / 0.008	✓
12	BOF	4,377	17,089	0.055 / 0.013	0.044 / 0.010	0.057 / 0.003	0.013 / 0.001	✓
13	Stack-based BOF_1	94	310	0.044 / 0.468	0.043 / 0.457	0.047 / 0.152	0.004 / 0.019	✓
14	Inter_DF	4,259	16,317	0.049 / 0.012	0.044 / 0.010	0.059 / 0.004	0.015 / 0.001	✓
15	Inter_UAF	4,252	16,331	0.047 / 0.011	0.045 / 0.011	0.067 / 0.004	0.022 / 0.002	✓
16	Stack-based BOF_2	94	310	0.046 / 0.489	0.044 / 0.468	0.056 / 0.181	0.012 / 0.056	✗

TABLE III: Experimental results on real-world-benchmarks.

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

[1] "IO" means integer overflow.
[2] "SBOF" means stack-based buffer overflow.

test cases, in which 61 are from CWE and 880 are extracted from the real world. Among the remaining 941 valid test cases, we detected 2 integer overflow in Wasm, 54 memory-safety bugs, in which 7 are use-after-free, 5 are buffer overflow, 2 are double-free, and 40 are stack-based overflow.

Summary: WASMDYPA detects 2 integer overflow bugs and memory-safety 54 bugs on our real-world systems, respectively, hence promising usefulness.

F. RQ3: Performance

To answer **RQ2** by investigating the performance and overhead WASMDYPA introduced, Table II (the 5th and 6th columns) presents the performance of WASMDYPA, including: 1) time for static instrumentation on Wasm (Instrumentation Time); 2) time for execution before and after instrumentation (EXE BI Time and EXE AI Time); and 3) time for dynamic analysis on WASMDYPA (Analysis Time). We ran each test case 100 rounds, then calculated the average. Experimental results demonstrated that WASMDYPA is efficient in detecting vulnerabilities in Wasm applications: the time spent on instrumentation is around 0.04 - 0.05 seconds for each case, with milliseconds varying from 0.978 to 0.011 per line of code; whereas the analysis time varies from 0.001 to 0.016 seconds for each case, with milliseconds varying from 0.003 to 0.176 per line of code. Moreover, the analysis time per line is much less than the execution time before and after instrumentation, thus the overhead of the dynamic analysis is negligible; and it is clearly demonstrated that selective instrumentation has a positive contribution to performance as an inverse trend with the time of per line of code. Hence, these experiments results demonstrated that WASMDYPA is efficient with negligible overhead.

Summary: WASMDYPA took around 50% time for its static instrumentation, while its dynamic analysis incurred negligible time-consuming with nearly no effect on Wasm execution time. The selective static instrumentation helped improve the the efficiency of the dynamic analysis significantly.

G. Case Study

To provide a more concrete understanding of the usefulness, we present two bugs detected by WASMDYPA, which respectively belong to *IO2BO* and *heap-based buffer overflow*. For more examples, we refer the readers to our source code repository.

IO2BO. Fig. 6 gives an integer overflow which further leads to an out-of-bounds access documented by CWE-190 [45]. This code snippet intends to allocate a table of size num_imgs, however as num_imgs grows large, the calculation determining the size of the list will eventually overflow and result in a very small list to be allocated instead. If the subsequent code operates on the list as if it were num_imgs long, it may result in out-of-bounds problems (*e.g.*, buffer overflow). The corresponding Wasm compiled from this C was shown in Fig. 6 either. It is notable that we simplified the origin Wasm to better demonstrate our example, `i32.mul` takes in

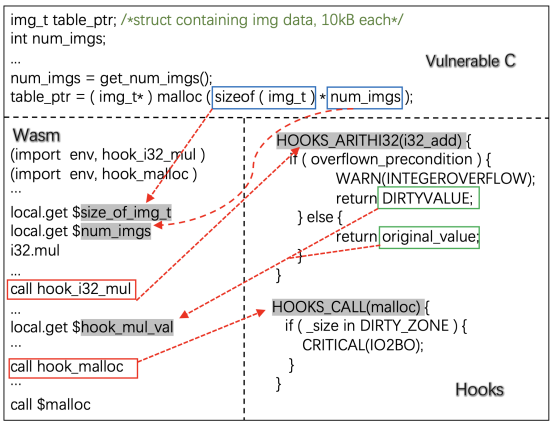


Fig. 6: CWE-190: Integer overflow or wraparound.

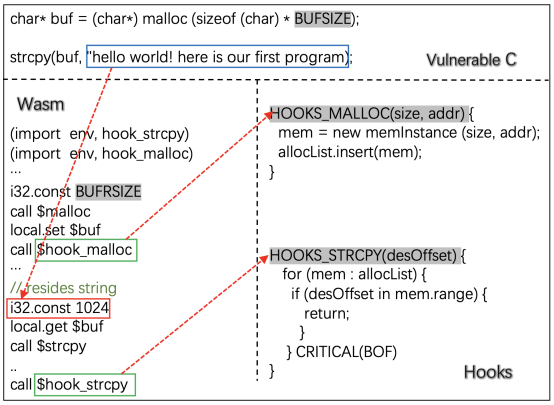


Fig. 7: CWE-122: Heap-based buffer overflow

the value of the size of image (*i.e.*, `size_od_img_t`) and the number of images (*i.e.*, `num_imgs`) to calculate the total space to be allocated. However, before allocating the memory, Wasm have to invoke the native hooks we attached to detect IO2BO: 1) the `hook_i32_mul`, takes in the multiplication result and two multipliers, to justify if an overflow happens. If the multiplication overflows, the hook will emit a warning and return the `DIRTYVALUE` to replace the original one. 2) The dirty value would be further propagated to the second hook, `hook_malloc`, before the allocation. The hook determines whether the income is a normal size or poisoned one, by checking the value is in the `DIRTY_ZONE`, if that is the case, WASMDYPA would emit critical messages to declare an IO2BO.

Heap-based BOF. Fig. 7 gives an example of heap-based buffer overflow documented by CWE-122 [63]. The buffer in this code snippet is allocated heap memory with a fixed size, but there is no guarantee the string will not exceed the size and cause an overflow. Hence, we instrument hooks around `malloc` and `strcpy` to collect the runtime memory information: 1) the `hook_malloc` takes in the allocated memory size and the buffer base, records it in a dedicated list; 2) the `hook_strcpy` takes in the offset after the string copy, looking up the dedicated list to check whether the offset exceeds the buffer. If that is the case, WASMDYPA would emit message to declare a BOF.

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.

Richer Wasm attribution support. WASMDYPA is confined to basic and commonly used instructions while there are still features in Wasm that we are not focused on currently, such as single instruction with multiple data (SIMD). As a result, WASMDYPA may not be effective in certain scenarios. At this point, we are planning to support as many attributions of Wasm as possible in the future to broaden the practicality of various situations.

More expressive memory detection model. WASMDYPA covers the common unsafe memory-related function calls in C compiled by clang provided by wasi-sdk. However, allocators in Wasm programs would diverge from the default allocator in clang which may result in different memory allocation behavior. Thus, by considering a more wide range of allocators, we may detect more subtle vulnerabilities. In this direction, we are planning to study the allocation strategies of self-designed and off-the-shelf allocators to supplement our detection algorithms.

Other vulnerabilities. Although WASMDYPA can effectively detect integer overflow and memory-related vulnerabilities in Wasm applications, it can also be extended to process more types of vulnerabilities. Especially, extending WASMDYPA with concurrency vulnerability checking capabilities will make it more comprehensive. In this direction, we can leverage the latest research progress [64] for concurrency security. We leave it an important future work.

VIII. RELATED WORK

In recent years, there are a significant amount of studies on Wasm security. However, the work in this paper stands for a novel contribution to this field.

Empirical security studies. Several empirical studies have been conducted on Wasm. Lehmann et al. [42] conducted an empirical studies on Wasm binary security. Musch et al. [65] characterized the prevalence of Wasm in the wild. Wang et al. [19] conducted an empirical study on Wasm runtimes to analyze bug root causes. Romano et al. [18] 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. Security is the key design goal of Wasm, thus analyzes for Wasm gain a lot of attention. Haas et al. proposed a small-step operational semantics and a type system to check the safety of stack-based operations for Wasm. Stiévenart et al. [66] proposed an information flow analysis

algorithm for Wasm programs and Lopes et al. [12] 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. [67] and Fu et al. [68] performed taint analysis on Wasm to track the propagation of data and detect possible input vulnerabilities. A key limitation of existing studies is that no comprehensive study on Wasm integer overflow. On the contrary, this work, for the first time, investigates and detects Wasm integer overflow vulnerability and their mitigations.

Dynamic analysis. Dynamic analysis has been extensively studied. Agrawal et al. [69] proposed dynamic slicing as a complementation to static slicing [70] with efficient debugging and testing capability. Newsome et al. [53] proposed dynamic taint analysis features fast automatic attack detection and filtering mechanisms for x86 binaries. Bond et al. [71] 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. Despite being a relatively new binary set, various dynamic analyses for WebAssembly have been proposed, including a taint analysis [72], a cryptomining detector [73] and a dynamic analysis framework [13]. Some of these analyses have been confined to browsers, such as conducting studies by modifying the V8 engine [74]. However, a portion of the existing Wasm dynamic analysis tools have scenario constraints and 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 runtimes, with efficiency and portability.

IX. Conclusion

This paper presented WasmDyPa, an infrastructure to secure integer overflow in Wasm program, which consists of three key components: fuzzing module, hooks instrumentation module, and dynamic analysis module: fuzzing module for instrumenting new execution paths to expose susceptible operations to generate the tainted input; a hooks instrumentation module to instrument hooks around potential sink sites to gather their runtime information; and an analysis module to detect runtime vulnerabilities based on dynamic analysis. We implemented a prototype for WasmDyPa and conducted systematic experiments with it. The experiment results demonstrated the effectiveness, efficiency, and usefulness of WasmDyPa. Overall, the work in this paper is a first comprehensive step towards securing the Wasm language vulnerabilities through dynamic ways, 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] "Cwe - cwe-680: Integer overflow to buffer overflow (4.11)," https://cwe.mitre.org/data/definitions/680.html.

[12] 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.

[13] 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.

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

[15] 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.

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

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

[18] 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.

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

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

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

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

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

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

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

[26] "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.

[27] "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.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] "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.

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] "Cwe - cwe-190: Integer overflow or wraparound (4.11)," https://cwe.mitre.org/data/definitions/190.html.

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

[47] "Openssh," https://www.openssh.com/.

[48] "Wasm-tools/crates/wasm-mutate at main · bytecodealliance/wasm-tools," https://github.com/bytecodealliance/wasm-tools.

[49] 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.

[50] 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.

[51] "Warf - webassembly runtimes fuzzing project," FuzzingLabs, May 2023.

[52] 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.

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

[54] 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.

[55] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones."

[56] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks."

[57] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *2008 IEEE Symposium on Security and Privacy (Sp 2008)*. Oakland, CA, USA: IEEE, May 2008, pp. 263–277.

[58] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. Orlando, FL, USA: IEEE, Dec. 2006, pp. 135–148.

[59] 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.

[60] "American fuzzy lop," Google, May 2023.

[61] "Bytecodealliance/wasm-micro-runtime: Webassembly micro runtime (wamr)," https://github.com/bytecodealliance/wasm-micro-runtime.

[62] "Webassembly/wabt: The webassembly binary toolkit," https://github.com/WebAssembly/wabt/tree/main.

[63] "Cwe - cwe-122: Heap-based buffer overflow (4.11)," https://cwe.mitre.org/data/definitions/122.html.

[64] 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.

[65] 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.

[66] 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.

[67] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," Jul. 2018.

[68] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," Feb. 2018.

[69] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990.

[70] M. Harman and R. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.

[71] M. D. Bond, N. Nethercote, and S. W. Kent, "Tracking bad apples: Reporting the origin of null and undefined value errors."

[72] Q. Stiévenart, "Wassail," Apr. 2023.

[73] 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.

[74] "V8 javascript engine," https://v8.dev/.