An Empirical Study of Vulnerabilities in WebAssembly Runtimes

Sara Baradaran

***Abstract*—WebAssembly runtimes provide a sandboxed environment that abstracts away the underlying hardware and consistently executes WebAssembly code. The presence of vulnerabilities in a runtime not only changes the code behavior at execution time but also can get exploited by attackers to gain control of systems relying on the runtime. Understanding the root causes of vulnerabilities in runtimes allows early detection and helps developers avoid similar vulnerabilities when developing analogous software. In this paper, we have collected and analyzed a set of 84 known vulnerabilities in WebAssembly runtimes both quantitatively and qualitatively. Through this study, we observed that many vulnerabilities share the same root causes. We classified the root causes into three different categories and investigated the distribution of each root cause. We also found that 10 reported vulnerabilities are still present and have not been patched by developers, which affects the security of systems that use vulnerable runtimes.**

***Index Terms*—WebAssembly, Runtime, Vulnerability Study, CVE Analysis**

I. Introduction

Since 2017, WebAssembly has emerged as a binary format and a portable target for compiling code written in high-level programming languages such as C/C++, Rust, and Go [1]. WebAssembly allows developers to execute computationally intensive tasks, such as image and video processing, cryptography, and data manipulation in near-native performance. While WebAssembly was initially designed for running code in web browsers, its applications have expanded beyond the web domain in blockchain [2]–[4], cloud computing [5]–[7], and Internet of Things (IoT) [8]–[10]. As WebAssembly is increasingly adopted for various applications, there is a growing ecosystem of WebAssembly runtimes. WebAssembly runtimes serve as sandboxed environments for executing WebAssembly applications. These runtimes can be broadly classified into two main categories: web browser engines and standalone runtimes. Browser-based engines, such as Google Chrome's V8 [11], Mozilla Firefox's SpiderMonkey [12], and Microsoft Edge's Chakra [13], incorporate WebAssembly support in their overall functionality. Specifically, they provide an integrated environment for running WebAssembly modules within web browsers. Table I demonstrates the most widely-used WebAssembly engines used in browsers.

Runtime	Source	Browser
V8	C++	Google Chrome
SpiderMonkey	C++/Rust/JavaScript	Mozilla Firefox
Chakara	C/C++/JavaScript	Microsoft Edge
JavaScriptCore	C/C++	Apple's Safari

TABLE I
Browser-based Runtimes for WebAssembly

On the other hand, standalone runtimes are independent implementations of WebAssembly engines designed to be used outside web browsers, catering to diverse application scenarios beyond the web, such as command-line tools, desktop applications, and embedded systems. There exist various projects on open-source platforms (e.g., GitHub), as shown in Table II, which are developed as standalone execution environments for WebAssembly [14], [15].

Typically, there exist three execution modes for running WebAssembly code, including just-in-time (JIT) compilation, ahead-of-time (AOT) compilation, and interpretation. Each standalone WebAssembly runtime may support one or more execution modes. Just-in-time (JIT) compilation is an execution technique that dynamically translates the bytecode or intermediate representation of a program into native machine code during runtime, just before it is getting executed. The on-the-fly compilation process optimizes the code for the specific hardware platform on which it is running by leveraging runtime information and profiling data. In contrast, ahead-of-time (AOT) compilation involves precompiling WebAssembly code into machine code before execution. The compiled native code is typically stored as a disk file and is loaded into memory when the WebAssembly code needs to be executed. By eliminating dynamic compilation during runtime, AOT compilation helps mitigate the performance overhead associated with JIT compilation. It allows for faster startup time and reduces memory consumption compared to JIT compilation.

Interpreters are another group of WebAssembly runtimes. A WebAssembly interpreter is a software component specifically designed to directly execute WebAssembly bytecode instead of first translating it into machine code. It reads the binary format of a WebAssembly module and sequentially executes the instructions it contains. An interpreter may rely on a set of functions written in high-level languages, which are invoked to emulate the desired functionality when a WebAssembly instruction needs to be executed.

Since WebAssembly's introduction, many vulnerabilities have been reported in WebAssembly runtimes, which enable attackers to take control of execution in these runtimes, thereby seriously harming systems and users. For example, approximately 60% of all Chrome's vulnerabilities discovered in the wild from 2021 to 2023 originated from memory corruption vulnerabilities within V8, Google Chrome's JavaScript and WebAssembly engine [16].

Runtime	Source	LoC	Stars	Active	Execution
GraalWasm	Java	66.5k	19.7k	Yes	?
Wasmer	Rust	225.9k	17.6k	Yes	AoT/JIT
Wasmtime	Rust	370.4k	14.3k	Yes	AoT/JIT
WasmEdge	C/C++	98.9k	7.8k	Yes	AoT/JIT
Wasm3	C/C++	15.0k	7.0k	Yes	Int
Lunatic	Rust	13.7k	4.5k	Yes	JIT
Wazero	Go	151.4k	4.5k	Yes	AoT/Int/?
WAMR	C/C++	172.9k	4.4k	Yes	AoT/JIT/Int
Lucet	Rust	30.2k	4.1k	No	AoT
Extism	Rust	6.4k	3.6k	Yes	?
WAVM	C/C++/Python	66.1k	2.6k	No	JIT
Life	Go	5.6k	1.7k	No	Int
Wasmi	Rust	68.3k	1.3k	Yes	Int
Wagon	Go	16.2k	902	No	Int
VMIR	C/C++	50.2k	651	No	JIT/Int
Wac	C/C++	3.6k	464	No	Int
aWsm	C/C++/Rust	386.3k	282	Yes	AoT
WasmVM	C/C++	11.9k	210	Yes	Int
Fizzy	C/C++	27.3k	206	No	Int
WasmKit	Swift	21.8k	$\leq$200	Yes	Int
py-wasm	Python	12.4k	$\leq$100	No	Int
Swam	Scala	20.6k	$\leq$100	No	Int
Warpy	RPython	3.2k	$\leq$100	No	JIT

TABLE II
STANDALONE RUNTIMES FOR WEBASSEMBLY

However, there is a notable absence of systematic studies on vulnerabilities specific to WebAssembly runtimes within our community. By comprehensively understanding detected vulnerabilities, developers can gain insights into common vulnerable code patterns, functions, and APIs, enabling them to avoid similar pitfalls when designing and implementing analogous software. Moreover, since many vulnerabilities often share the same root causes, delving into the root causes of known vulnerabilities can unveil new security vulnerabilities. Understanding the security issues of runtimes can also notify users about security concerns of the underlying runtimes they rely on.

In this research, we conduct a thorough assessment of known vulnerabilities in WebAssembly runtimes to identify recurring patterns and root causes, with the ultimate goal of improving the security posture of WebAssembly-based systems. Through systematic analysis of vulnerabilities, we aim to provide actionable recommendations and guidelines to developers, enabling them to build more robust and secure WebAssembly applications and runtimes. This study answers the following research questions.

RQ1: What are the root causes of vulnerabilities in WebAssembly runtimes and how they are distributed? Root causes help researchers gain insight into the nature of vulnerabilities. To answer this question, we first categorized root causes for vulnerabilities based on a systematic process, and three root causes are identified. We then analyze the root cause distribution of these vulnerabilities.

RQ2: How many legacy vulnerabilities have already been patched? We tried to reproduce 62 vulnerabilities using proof of concept (PoC) files provided by people who reported vulnerabilities for the first time. We found noticeable variations in the degree of responsibility within the developer's community when fixing vulnerabilities detected in different runtimes. Furthermore, we observed that out of 62, 10 vulnerabilities are still alive in the runtimes.

RQ3: What types of weaknesses are more prevalent? We explored the prevalence of different types of weaknesses within WebAssembly runtimes. By investigating the Common Weakness Enumeration (CWE) categories associated with these vulnerabilities, we seek to gain insights into the patterns of weakness occurrences. This can provide information for prioritizing security efforts, enhancing mitigation strategies, and guiding the development of more robust runtimes.

RQ4: According to the results of this study, what recommendations can we make for developers of runtimes to improve the security and robustness of their products?

II. DATA COLLECTION

A. Selection of WebAssembly Runtimes

Besides four widely-used browser-based engines that support WebAssembly code execution, we inspect WebAssembly runtimes projects on GitHub using the curated awesome-wasm lists [14], [15] that include more than 30 WebAssembly runtimes currently available. As shown in Table II, these standalone runtimes are written in different programming languages, which provide different levels of language safety. A group of runtimes are actively maintained by developers, while the remaining are no longer maintained (the rows colored in gray). If the most recent commit in a project's repository occurred over a year ago, we assume the project has reached the end of its lifecycle. In some cases, the project's developers have explicitly mentioned that the project is no longer under maintenance. For example, developers of Lucet switched focus to working on the Wasmtime engine in mid-2020. They asserted that all of Lucet's features are already ported to Wasmtime.

In this study, we focus on sufficiently mature runtimes, whether they are now active or not. Specifically, we pruned out small projects with less than 3k LoC, which are developed as experimental runtimes and only support a minimal specification of WebAssembly, since there is no known vulnerability for them either as a CVE or a GitHub issue. As a result, such projects mentioned in awesome-wasm lists have not been included in Table II.

B. Collection of Vulnerabilities in WebAssembly Runtimes

To investigate the characteristics of WebAssembly runtime vulnerabilities, we first crawled 114 Common Vulnerabilities and Exposures (CVEs) from the National Vulnerability Database (NVD) using the keywords "WebAssembly" and "Wasm". Then, we pruned 53 CVEs out of 114, as these vulnerabilities are associated with non-runtime WebAssembly tools (e.g., Binaryen, WABT) designed for compiling, optimizing, and debugging WebAssembly code.

Besides using the NVD dataset, we looked for issues and pull requests whose content included keywords such as "vulnerability" and "security". We used the GitHub REST API [17] to build the crawler and retrieved 123 reported vulnerabilities across XX different WebAssembly runtimes.

III. Study of Vulnerabilities in WebAssembly Runtimes

A. RQ1: Vulnerability Root Causes

We investigate vulnerability information to identify the types of root causes among the issues. Specifically, for each reported vulnerability available in the NVD database, we read the vulnerability description alongside the patch commit and conversation on the GitHub issue page (if available) to find out the underlying reasons that resulted in the vulnerability. Through this process, we found the following three root causes for vulnerabilities in WebAssembly runtimes.

Missing of WebAssembly input validation. WebAssembly input validation refers to the process of verifying the correctness and safety of WebAssembly modules before execution. It involves checking the structure, types, and instructions within the Wasm module to ensure they adhere to the WebAssembly specification [18]. In this study, we observed that many vulnerabilities have been reported in WebAssembly runtimes, which all can be triggered by passing an invalid input file to these runtimes. Further analysis of such vulnerabilities shows that they are introduced since a group of runtimes does not involve a systematic mechanism for validating input modules before and during execution. For example, Wasm3 currently assumes that input modules (`.wasm` files) are already validated. Though this runtime performs many checks before executing the input, the Wasm3 validation process is incomplete yet. As a result, Wasm3 gets various fuzzing-detected vulnerabilities, the majority of which are produced by invalid WebAssembly inputs. Moreover, we observed that several runtimes (e.g., WAVM, WAMR) validate WebAssembly inputs at the present time, while in the early stages of the development of these runtimes, input validation was not among the developers' priorities. Therefore, the GitHub repositories of these runtimes are full of issues referring to legacy vulnerabilities that are no longer alive in the codebase. In the following, we discuss three cases where we observed a vulnerability could be triggered by an invalid WebAssembly input.

a) Control Stack Handling: We identified several vulnerabilities in the early versions of WAVM, all of which stem from the lack of control stack checking during the execution of an invalid WebAssembly input. In WebAssembly, control flow instructions (e.g., `if`, `else`, `end`, `loop`, and `block`) are used to structure the control flow of programs. These instructions push and pop control frames onto a control stack as they faced during execution. When executing a WebAssembly module, the control stack keeps track of nested control structures and their associated state. Each time a control frame is encountered, it is pushed onto the control stack, and when the control structure is completed, the corresponding frame is popped. If a given WebAssembly input violates the WebAssembly specification, it may contain incorrect or mismatched control flow instructions. These violations might cause the control stack to become empty at a certain point before reaching the end of execution, thereby bringing about crashes. Moreover, when ending the execution, the control stack should be empty, indicating that all control structures, including function calls and nested control flow constructs, have been properly completed and returned. However, in the case of an invalid WebAssembly module, the control stack may not be empty at the end of execution, which might result in undefined behaviors.

```
1 - //Advance until a recovery point
2 - while(!isRecoveryPointChar(*nextChar)) {++nextChar;}
3 + //Advance until a recovery point or the end of the string
4 + const char* stringEnd = string + stringLength;
5 + while(nextChar < stringEnd &&
6 +            !isRecoveryPointChar(*nextChar)) {++nextChar;}
```

Fig. 1. Vulnerable code snippet in WAVM's lexer

b) Error Recovery Handling: Error recovery in a lexer refers to the technique employed to handle errors that arise during the scanning phase of the input source code. If the lexer encounters an invalid token in the source code, it typically raises an error and halts the parsing process. By implementing error recovery mechanisms, the lexer tries to identify the point of error and takes appropriate actions to continue processing the remaining code. Thus, it can provide more informative error messages to developers while still attempting to analyze as much of the code as possible. WAVM lexer component supports error recovery. If it encounters an invalid token, it will advance until it reaches a recovery point. However, there was a bug in implementing the lexer error recovery where the lexer continued to parse without considering the end of the input. As a result, it may cause out-of-bounds memory access when loading an invalid WebAssembly input. The commit associated with vulnerable source code and its patch is demonstrated in Fig. 1. The patched version of the lexer continues parsing until either it reaches a recovery point or the end of the input string.

c) Function Call Handeling: Function declarations provide a way to declare the existence and signature of a function without providing its actual implementation. On the other hand, function definitions include the implementation code for the declared functions. If a WebAssembly module includes function declarations without a corresponding function definition, it should be considered an invalid input. CVE-2018-16769 [19] is a known vulnerability introduced in WAVM since in early versions of this runtime, the loader does not reject an input containing such a declared function without definition, which subsequently leads to buggy behavior when calling the aforementioned function.

Implementation bugs for WebAssembly new features. Since the advent of WebAssembly, several proposals have been proposed to enhance and expand WebAssembly's capabilities. We found that a class of vulnerabilities was introduced in WebAssembly runtimes due to the bugs in implementing the newly proposed features. In the following, we review some of these features, their implementation intricacy, and the vulnerabilities they caused.

```
  #[inline]
  fn into_abi(self, store: &mut StoreOpaque) -> Self::Abi {
    if let Some(x) = self {
      let abi = x.inner.as_raw();
      unsafe {
-       store.insert_vmexternref(x.inner);
+       let mut store = AutoAssertNoGc::new(store);
+       store.insert_vmexternref_without_gc(x.inner);
      } abi
    } else { ptr::null_mut() }
  }
- pub unsafe fn insert_vmexternref(&mut self, r: VMExternRef) {
-   self.externref_activations_table.insert_with_gc(r, &self.modules)
+ pub unsafe fn insert_vmexternref_without_gc(&mut self, r: VMExternRef) {
+   self.externref_activations_table.insert_without_gc(r);
  }
+ // Insert a reference into the table, without ever performing GC.
+ #[inline]
+ pub fn insert_without_gc(&mut self, externref: VMExternRef) {
+   if let Err(externref) = self.try_insert(externref)
+     self.insert_slow_without_gc(externref);
+ }
+ #[inline(never)]
+ pub fn insert_slow_without_gc(&mut self, externref: VMExternRef) {
+   self.over_approximated_stack_roots.insert(VMExternRefWithTraits(externref));
+ }
```

Fig. 2. Vulnerable code snippet in Wasmtime's externref.rs source file

d) Reference Types: This proposal adds support for reference types in WebAssembly, which aims to enable more efficient communication between the host and the Wasm module. Without the reference types, WebAssembly cannot take or return references to the host's objects (e.g., DOM node on a web page). Instead, one can store the host objects in a side table and refer to them by index. However, this adds an extra indirection, and implementing the side table requires cooperating glue code on the host side. The glue code is also troublesome since it is host-specific and outside the Wasm sandbox, diluting Wasm's safety guarantees. In this way, if one intends to use the Wasm module on the web, in a Rust host, and in a Python host, three separate glue code implementations are needed, causing WebAssembly to become less attractive as a universal binary format. Thus, the value type *externref* removes the need for glue code to interact with host references.

The proposal for reference types has three main parts. First, the new *externref* type that represents a reference to a host object. Second, an extension to WebAssembly tables, allowing them to hold *externrefs* in addition to function references. Third, new instructions for manipulating tables and their entries. With these new capabilities, Wasm modules can directly talk about host references rather than requiring external glue code running in the host. The *externref* should not violate the WebAssembly's sandboxing properties. Therefore, it is supposed to be opaque and unforgable.

Being opaque means that a WebAssembly module cannot observe an *externref* value's bit pattern. Since *externref* is completely opaque from the module perspective, the only way to use it is to send an *externref* back to the host as an argument of an imported function. Passing a reference to a host object into a Wasm module also does not reveal any information about the host's address space and the layout of host objects within it. The *externref* is also unforgable, meaning that a Wasm module cannot create a fake host reference out of thin air. An *externref* cannot be dereferenced by the module, and the module cannot directly access or modify the data behind the reference. Indeed, the module cannot even be sure which kind of data is being referenced. It can only return either a reference we already gave it or a null reference.

CVE-2021-39216 [20] refers to a vulnerability in Wasmtime that is triggered when the host passes multiple *externrefs* to a WebAssembly instance at the same time. If Wasmtime's VMExternRefActivationsTable[1] became full after passing in the first *externref*, then a garbage collection will occur when the second *externref* is passed. However, the first *externref* is not protected from being collected and could become free. Later, when control is given to the WebAssembly content, it could still try to use the first *externref* even though it has been freed, thereby resulting in a use-after-free vulnerability. Fig. 2 shows a part of the commit patching this vulnerability. According to this patch, new functions are added which insert the new *externref* into the table without performing garbage collection.

e) SIMD (Single Instruction, Multiple Data): SIMD is a proposal that aims to introduce native support for vector operations in WebAssembly, enabling the parallel process of data. SIMD instructions are a specific class of instructions that exploit data parallelism in applications by simultaneously performing the same operation on multiple data elements. Computationally intensive applications such as audio/video code and image processors take SIMD instructions to accelerate performance. The high-level goal of the WebAssembly SIMD proposal is to introduce vector operations to the WebAssembly specification in a way that guarantees portable performance.

This proposal defines a portable, performant subset of SIMD operations available across most modern architectures. The current proposal is limited to standardizing fixed-width 128-bit SIMD operations. It introduces a new v128 value type, used to represent different types of packed data, and a number of new operations that operate on this type (on packed data). To determine these operations, some criteria are considered. First, the operations should be well-supported across multiple modern architectures. Second, performance wins should be positive across multiple relevant architectures within an instruction group. Finally, the chosen set of operations should minimize performance cliffs if any.

CVE-2023-41880 [21] has been assigned to a vulnerability in the Wasmtime, which is triggered when using the `i64x2.shr_s` instruction (stands for "shift right signed") with a constant shift amount larger than 32 bits on x86-64 platforms. The `i64x2.shr_s` WebAssembly instruction performs a right shift operation on each lane of a 128-bit vector

[1]A data structure used to manage the lifetime of external references passed to WebAssembly modules.

Runtime	#Vulnerability	#Available PoC	#Alive	#Fixed	#CVE
Wasmtime	15	0	0	0	15
Wasm3	16	12	8	4	9
WAMR	42	41	0	41	2
WAVM	11	9	2	7	9

TABLE III
RESULTS OF VULNERABILITY REPRODUCTION IN WEBASSEMBLY STANDALONE RUNTIMES

of two signed 64-bit integer values (i64). This instruction takes two operands: a 128-bit vector of two i64 values (referred to as a lane) and a 64-bit scalar value (referred to as a shift amount). It shifts each lane of the vector to the right by the number of bits specified by the shift amount, considering the sign of the values. This bug causes the instruction to produce an incorrect result, bringing about an off-by-one error (OBOE) vulnerability[2] The vulnerability happens due to a miscompilation of `i64x2.shr_s` instruction by the Wasmtime's Cranelift code generator [22]. We also found other vulnerabilities in Wasmtime, coming from incorrect machine code generation of instructions in WebAssembly SIMD proposal.

Bugs in Code Generator Components. Those WebAssembly runtimes that support the JIT or AOT compilation use code generators in the backend to translate WebAssembly into machine code (just) before execution. The code generation components are usually developed by other teams and the runtimes developers only use them without knowing the implementation details. Cranelift [22] and LLVM [23] are two widely-used code generators employed by many runtimes. Indeed, vulnerabilities in these modules can make the entire relying runtime vulnerable.

As mentioned, bugs in the code generators caused the miscompilation of some instructions proposed in the WebAssembly SIMD proposal. However, we found that code generation bugs are not limited to the SIMD instructions. For example, CVE-2022-31169 [24] refers to a vulnerability according to which, in Cranelift, the translation rules for constants did not take into account whether sign or zero-extension should happen. Thus, it might result in an incorrect value being placed into a register when a division is encountered.

Another example is CVE-2023-26489 [25], which was introduced due to a bug in Cranelift on x86-64 targets where address-mode computation mistakenly calculates a 35-bit effective address instead of WebAssembly's defined 32-bit effective address. It means that, with default code generation settings, a load/store operation could read/write addresses from/to a ≈34GB address space instead of a 4GB linear memory[3], which can be exploited by malicious code.

To show the distribution of each root cause among vulnerabilities, Table IV summarizes the number of vulnerabilities associated with each root cause. It is worth noting that the last column shows the number of vulnerabilities that have been introduced since some coding practices or error-handling mechanisms are overlooked. For instance, the absence of range length checking for buffers or arrays. These vulnerabilities have no specific pattern and might be introduced in every component due to the lack of secure coding practices.

[2]The low 32-bits of the second lane of the vector are derived from the low 32-bits of the second lane of the input vector instead of the high 32-bits.

[3]Wasmtime's default sandbox settings indeed provide up to 6GB of protection from the base of linear memory (insted of 4GB) to guarantee that any memory access in that range will be semantically correct

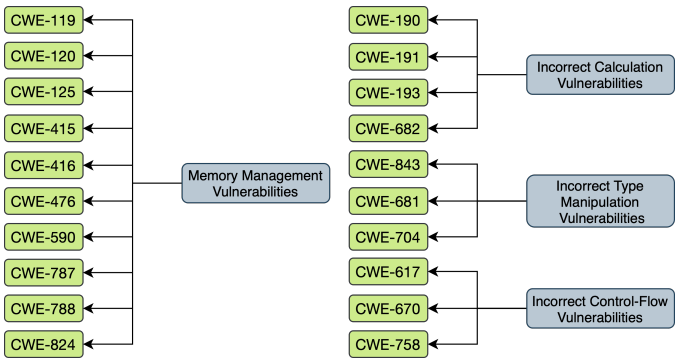

Fig. 3. Categorizing CWEs associated with vulnerabilities in WebAssembly runtimes into broader classes

Finding 1. WebAssembly input validation within a runtime seems an essential part of a secure and robust execution environment. This validation process involves different aspects from checking input tokens, code syntax, and types to handling control stack during runtime.

B. RQ2: Vulnerability Persistency

In this research question, we aim to investigate the current status of legacy vulnerabilities within runtimes and evaluate the extent to which they have been addressed. Specifically, we seek to determine the number of vulnerabilities that remain unresolved and compare it to the number that have been successfully fixed. This analysis provides insights into the security measures implemented by different runtimes. Our observations indicate that the level of attention given to vulnerabilities varies among developers of different runtimes.

To answer this research question, we tried to reproduce each vulnerability using the proof of concept (PoC) exploit provided by an individual who first reported that vulnerability. Unfortunately, we do not have access to the exploit for all vulnerabilities since a group of them have been reported directly to developers and instructions for reproduction are not publicly available. Overall, for 62 vulnerabilities we found the PoC exploits and tried to reproduce the vulnerabilities both in vulnerable versions and the latest version of runtime to first make sure if a vulnerability is not fake and second to understand what portion of vulnerabilities are still alive in the most recent version of runtimes.

Out of 62 vulnerabilities, we could reproduce all of them in the legacy versions of runtimes in which vulnerabilities are reported. When we tried to reproduce vulnerabilities using the most recent version of runtimes, we observed that 10 vulnerabilities are still alive. A large portion of these vulnerabilities are for Wasm3, which is known as a lightweight WebAssembly interpreter that focuses on quick startup. As mentioned earlier, this runtime does not provide a systematic mechanism for input validation. As a result, we found that

Runtime	Improper Handling of New Features		Incomplete Input Validation	Bugs in Code Generators	Implementation Bugs
	Reference Types	SIMD			
Wasmtime	5	3	0	5	2
Wasm3	0	0	15	0	1
WAMR	0	0	40	0	2
WAVM	0	0	9	0	2

TABLE IV
DISTRIBUTION OF ROOT CAUSES AMONG VULNERABILITIES

```
sara@sara:~/wasm3/build$ ./wasm3 CVE-2022-44874.wasm
AddressSanitizer:DEADLYSIGNAL
=================================================================================
==3143889==ERROR: AddressSanitizer: SEGV on unknown address 0x63100003b188
      (pc 0x560ac50f0df2 bp 0x62d000000490 sp 0x7fff96387c80 T0)
==3143889==The signal is caused by a READ memory access.
    #0 0x560ac50f0df2 in op_CallIndirect (~/wasm3/build/wasm3+0x59df2)
    #1 0x560ac50f8ac5 in m3_CallArgv (~/wasm3/build/wasm3+0x61ac5)
    #2 0x560ac50bbc16 in repl_call (~/wasm3/build/wasm3+0x24c16)
    #3 0x560ac50b901e in main (~/wasm3/build/wasm3+0x2201e)
    #4 0x7fddldc29d8f in __libc_start_call_main ../sysdeps/nptl/
       libc_start_call_main.h:58
    #5 0x7fddldc29e3f in __libc_start_main_impl ../csu/libc-start.c:392
    #6 0x560ac50bbl14 in _start (~/wasm3/build/wasm3+0x24114)

SUMMARY: AddressSanitizer: SEGV (~/wasm3/build/wasm3+0x59df2) in op_CallIndirect
==3143889==ABORTING

sara@sara:~/wasm3/build$ wasm2wat CVE-2022-44874.wasm  -o CVE-2022-44874.wat
0000030: error: invalid utf-8 encoding: import module name
```

Fig. 4. Reproducing CVE-2022-44874 in the latest version of Wasm3 using an invalid WebAssembly input

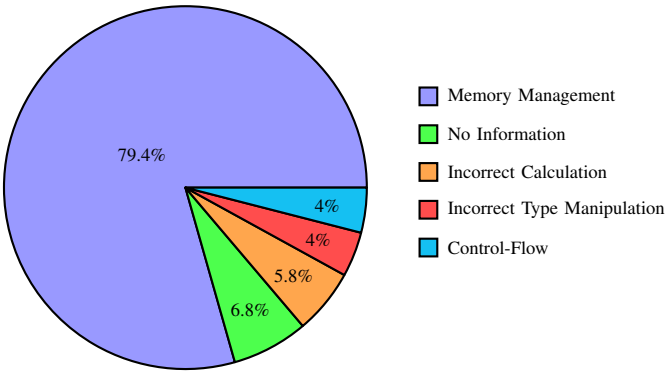

Fig. 5. Distribution of weaknesses among vulnerabilities

executing an invalid WebAssembly input using Wasm3 is most likely to trigger a vulnerability in this runtime. For example, Fig. 4 shows that we could successfully reproduce CVE-2022-44874 [26] by using an invalid WebAssembly input in the latest version of Wasm3. Table III summarizes our results to answer RQ2. The columns from left to right indicates the total number of vulnerabilities collected for each runtime, the number vulnerabilities for which exploit files are available, the number of vulnerabilities still alive in the codebase, the number of vulnerabilities that are patched, and the number of official vulnerabilities to which CVE ID are assigned. It is worth noting that for each runtime (#Vulnerability - #CVE) shows the number of vulnerabilities that are reported through GitHub issues, but there are not available in NVD database.

Finding 2. There are known vulnerabilities in WebAssembly runtimes that have been left without patches. These vulnerabilities can put the security of runtimes and users who use them at risk if they are unaware, as their exploits are publicly available. Out of 62 vulnerabilities for which exploits are available, we observed that 10 (≈16%) are still alive.

C. RQ3: Vulnerability Weakness Types

While understanding the distribution of CWEs among vulnerabilities is valuable, merely focusing on the exact type of CWE for each vulnerability may not provide significant insights. Instead, we make a shift in granularity to investigate the distribution of weaknesses within broader classes. In Fig. 3, we classified CWEs into broader and less granular classes. For example, instead of considering CWE-190: integer overflow or wraparound, CWE-191: integer underflow (wrap or wraparound), CWE-193: off-by-one error, and CWE-682: incorrect calculation in isolation, we place all these weaknesses in a single class and label them as incorrect calculation vulnerabilities as all of them results in incorrect values being computed. Here, we consider the following five class of weaknesses that each class is indeed obtained by merging several CWEs as shown in Fig 3.

- **Memory Management Vulnerabilities.** weaknesses that arise from improper handling or manipulation of memory resources.
- **Incorrect Calculation Vulnerabilities.** weaknesses that arise from errors or inaccuracies in mathematical or computational calculations.
- **Incorrect Type Manipulation Vulnerabilities.** weaknesses that arise from improper handling, casting, or manipulation of data types.
- **Incorrect Control-Flow Vulnerabilities.** weaknesses that arise due to control flow path that does not reflect the algorithm that the path is intended to implement, leading to incorrect behavior any time this path is navigated.
- **Vulnerabilities with Insufficient Information.**

For 84 vulnerabilities analyzed in four WebAssembly standalone runtimes, Fig. 5 depicts how prevalent is each class of weaknesses among these vulnerabilities.

Finding 3. Memory management vulnerabilities are the most prevalent weakness type among vulnerabilities in WebAssembly runtimes. Also, 85.2% vulnerabilities arise from memory management and incorrect calculation vulnerabilities.

IV. RELATED WORK

Analyzing Bugs in WebAssembly Runtimes. Some researchers have recently studied bugs in WebAssembly runtimes [27], [28]. Wang et al. [27] conducted an empirical analysis of more than 800 reported bugs across four WebAssembly runtimes. They analyzed bugs regarding their root

causes, symptoms, bug-fixing time, and the number of files and lines of code involved in the bug fixes. Through this study, they found that Incorrect Algorithm Implementation is the most prevalent root cause, and program crash is the most common symptom of bugs present in WebAssembly runtimes. Moreover, at the median, the bug-fixing time is less than 14 days. They also observed that more than half of bug fixes involve only one file, and on average, less than 40 lines of code need to be changed for bug fixes.

Likewise, Zhang et al. [28] studied 311 bugs in three WeAssembly standalone runtimes. They categorized these bugs into 31 categories and summarized their common fix strategies. They observed that these 311 bugs often have specific patterns and share similarities. Hence, Zhang et al. developed a pattern-based bug detection framework to identify bugs in WeAssembly runtimes. This framework specifically generates test cases specific to each bug category. For example, bug-triggering file operations include renaming, moving, counting, and mapping. Hence, the framework tests if a WebAssembly runtime can correctly rename a file or report error information when the file does not exist. Otherwise, it shows the presence of a bug related to file-handling operations.

Bugs primarily affect functionality and usability, while vulnerabilities directly impact the security of software systems. Though bugs are worth considering, addressing vulnerabilities is typically of higher priority due to their potential for confidentiality, integrity, and availability breaches. In this research, we target analyzing WebAssembly runtime vulnerabilities.

Analyzing Vulnerabilities in Virtual Machines. Some other researchers have targeted vulnerability studies in virtual machines (VMs) [29]. In [29], Yilmaz et al. assert that inconsistent, erroneous, or ambiguous vulnerability information hinders the identification and resolution of vulnerabilities in web-based virtual machines. For example, a large portion of ActionScript vulnerabilities are ambiguously classified as *memory corruption* vulnerabilities by the CVE database. However, a deeper investigation shows they can be classified as stack overflow, heap overflow, use-after-free, double-free, and integer overflow vulnerabilities. Specifically, the authors reclassified ActionScript vulnerabilities labeled as generic *memory corruption* and *unspecified* into one of the more fine-grained sub-classes. They also proposed Inscription, a method for transforming Adobe Flash binary code, which enables protection against different categories of Flash vulnerabilities without modifying vulnerable Flash VMs.

V. Conclusion

In this study, we collected a set of 184 known vulnerabilities in WebAssembly runtimes. We analyzed 84 vulnerabilities in four different WebAssembly runtimes to understand their root causes. Based on our analysis, incomplete input validation is the most common root cause for reported vulnerabilities. Also, we tried to reproduce those vulnerabilities with available exploits to investigate what portion of them are still alive in the runtimes. We observed that out of 62 vulnerabilities with available exploit, 10 vulnerabilities are still alive in runtimes. We found that more than 85% vulnerabilities arise from incorrect memory management and incorrect calculations.

Authors Contributions. Sara Baradaran: Analyzing and reproducing vulnerabilities in Wasm3, WAVM, WAMR, and Wasmtime, writing and editing paper draft, collecting related work papers, and collecting CVEs.
Jungkyu Kim: Programming for crawler, collecting data from GitHub issues, and analyzing vulnerabilities in Wasmtime.
Ritu Pravakar: Collecting related work papers.
Yutian Yan: Analyzing vulnerabilities in FireFox (incomplete).

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, jun 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[2] 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*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 703–715. [Online]. Available: https://doi.org/10.1145/3533767.3534218

[3] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "EOSAFE: Security analysis of EOSIO smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1271–1288. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu

[4] L. Quan, L. Wu, and H. Wang, "Evulhunter: Detecting fake transfer vulnerabilities for eosio's smart contracts at webassembly-level," 2019.

[5] M. N. Hoque and K. A. Harras, "Webassembly for edge computing: Potential and challenges," *IEEE Communications Standards Magazine*, vol. 6, no. 4, pp. 68–73, 2022.

[6] P. Mendki, "Evaluating webassembly enabled serverless approach for edge computing," in *2020 IEEE Cloud Summit*, 2020, pp. 161–166.

[7] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Webassembly as a common layer for the cloud-edge continuum," in *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*, ser. FRAME '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3–8. [Online]. Available: https://doi.org/10.1145/3526059.3533618

[8] E. Wen and G. Weber, "Wasmachine: Bring iot up to speed with a webassembly os," in *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2020, pp. 1–4.

[9] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 261–272. [Online]. Available: https://doi.org/10.1145/3498361.3538922

[10] B. Li, W. Dong, and Y. Gao, "Wiprog: A webassembly-based approach to integrated iot programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[11] "V8: Google chrome's javascript and webassembly engine." [Online]. Available: https://v8.dev

[12] "Spidermonkey: Mozilla's javascript and webassembly engine." [Online]. Available: https://spidermonkey.dev

[13] "Chakracore: Microsoft edge's javascript and webassembly engine." [Online]. Available: https://github.com/chakra-core/ChakraCore

[14] "Awesome webassembly runtimes." [Online]. Available: https://github.com/jcbhmr/awesome-webassembly-runtimes?tab=readme-ov-file

[15] "Awesome webassembly runtimes." [Online]. Available: https://github.com/appcypher/awesome-wasm-runtimes

[16] "The V8 Sandbox." [Online]. Available: https://v8.dev/blog/sandbox

[17] "GitHub REST API." [Online]. Available: https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

[18] "WebAssembly Specification." [Online]. Available: https://webassembly.github.io/spec/core
[19] "CVE-2018-16769." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2018-16769
[20] "CVE-2021-39216." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-39216
[21] "CVE-2023-41880." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-41880
[22] "Cranelift." [Online]. Available: https://cranelift.dev
[23] "LLVM WebAssembly backend." [Online]. Available: https://github.com/llvm/llvm-project/tree/main/llvm/lib/Target/WebAssembly
[24] "CVE-2022-31169." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-31169
[25] "CVE-2023-26489." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-26489
[26] "CVE-2022-44874." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2022-44874
[27] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of webassembly runtime bugs," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2023, pp. 355–366.
[28] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu, "Characterizing and detecting webassembly runtime bugs," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, dec 2023.
[29] F. Yilmaz, M. Sridhar, A. Mohanty, V. Tendulkar, and K. W. Hamlen, "A fine-grained classification and security analysis of web-based virtual machine vulnerabilities," *Computers & Security*, vol. 105, p. 102246, 2021.