SwFuzz: Structure-Sensitive WebAssembly Fuzzing

Jiashui Wang[1], Ziyi Guo[2], Xinlei Ying[3], Peng Qian[4], Yan Chen[2]
[1] Zhejiang University & Ant Group; [2] Northwestern University; [3] Ant Group; [4] Zhejiang University
12221251@zju.edu.cn; xinlei.yxl@antgroup.com; messi.qp711@gmail.com; {ziyi.guo,ychen}@northwestern.edu

***Abstract*—WebAssembly (WASM) has rapidly emerged as a ubiquitous target for web browsers, server-side applications, and blockchain platforms, with promising performance and portability. As WASM grows in popularity, ensuring its security and resilience becomes paramount. However, traditional fuzzing approaches struggle to detect potential security vulnerabilities in existing WebAssembly runtimes due to their lack of perception of the WASM file structure.**

In this paper, we introduce SwFuzz, a dedicated fuzzing framework tailored for WASM binaries. SwFuzz integrates comprehensive structure-sensitive policies that capture the nuances and intricacies within the WASM binaries. Our proposed fuzzing framework not only identifies vulnerabilities present in conventional binaries but also emphasizes the detection of WASM-specific bugs that have previously gone unnoticed. Experimental results demonstrate that SwFuzz has discovered numerous new bugs, with 17 CVEs being assigned, underscoring the importance of a specialized fuzzing framework for evolving platforms like WASM. Our findings also highlight the critical requirement for a proactive approach to securing the WASM landscape.

***Index Terms*—WebAssembly, Fuzzing, Vulnerability Detection**

I. Introduction

In the rapidly evolving landscape of the Internet, the emergence of Web3.0 represents a significant paradigm shift, empowering decentralized applications (dApps), blockchain systems, and other domains [1]. At the heart of this transformation lies WebAssembly (WASM), a cutting-edge technology that is redefining the boundaries of development and deployment of applications [2]. WASM provides a compact binary format that enables near-native performance for web applications, making it a cornerstone for a wide range of applications in the Web3.0 ecosystem. From improving the efficiency of decentralized applications to enabling complex computation tasks in web browsers without compromising on speed or security, WASM is emerging as a key technology across multiple domains. Its universal compatibility across different web platforms ensures that it is not only a powerful tool for developers but also a foundational element for the future of web innovation.

In particular, WASM is instrumental in enabling smart contracts to execute with higher efficiency and security, thus improving the scalability and interoperability of blockchain networks [3]. Within web browsers, WASM's ability to run code at near-native speeds has revolutionized the user experience, allowing complex applications such as gaming, graphics rendering, and scientific simulations to run smoothly without the need for plugins or external software [4]. Moreover, WASM's platform-agnostic nature facilitates its deployment in serverless computing, mobile applications, and Internet of Things (IoT) devices [5], demonstrating its potential to drive innovation and improve performance across various computing landscapes. This widespread applicability of WASM underscores its importance in the current technology era and signifies its role in shaping digital infrastructure and applications.

When it comes to existing WebAssembly fuzzing techniques, it is important to recognize the advancements and limitations that have characterized the field thus far. Traditional fuzzing methods applied to WASM, such as Fuzzm [6] and WasmFuzzer [7], have laid the groundwork for identifying vulnerabilities and ensuring the robustness of WASM applications. They mainly focus on generic fuzzing strategies, applying broad and rigid test inputs without a deep understanding of the unique WASM structure. While effective to a degree, these approaches often fall short in efficiently navigating the complex and dense landscape of WASM's binary format, resulting in suboptimal coverage and the potential oversight of deep-seated vulnerabilities.

Moreover, many existing fuzzing solutions [8]–[10] lack the specificity to detect and exploit the structure nuances of WASM files, resulting in a higher rate of false positives and an inefficient allocation of fuzzing resources. While these tools can perform broad security assessments, they often fail to accurately identify structure-specific vulnerabilities that adversaries may exploit in real-world scenarios. Additionally, the adaptability of these tools to the evolving standards and features of WASM has been another point of contention. This leaves security gaps that are increasingly exploited as technology advances.

To address these problems, we propose SwFuzz, a novel approach that leverages the inherent structure of WASM files to provide more efficient and targeted fuzzing for WASM. Unlike its predecessors, SwFuzz is designed to deeply understand the binary composition and execution flow of WASM modules, enabling it to generate test inputs that are much more likely to uncover vulnerabilities hidden in deep state space. By focusing on the WASM structure, SwFuzz is able to reduce the noise of irrelevant test cases and concentrate on the critical paths that traditional fuzzing tools might overlook. Such a structure-sensitive strategy not only increases the effectiveness of fuzzing campaigns but also significantly reduces the time and computation resources required for security assessments of WASM applications. We anticipate SwFuzz to set a new standard in the field of WASM fuzzing, addressing the shortcomings of existing methods and pushing the boundaries of WebAssembly security testing.

Contributions. The key contributions of our work can be

summarized as follows:

- *Design and Implementation.* We present SwFuzz, a novel fuzzing framework specifically tailored for the structure of WASM files. This framework represents a significant advancement in detecting vulnerabilities in WASM runtimes by leveraging a deep understanding of the WASM binary format.
- *Comprehensive Evaluation.* SwFuzz has been rigorously tested across various WASM runtimes, demonstrating its effectiveness in identifying vulnerabilities and showcasing its advantages over existing fuzzing methods through detailed benchmarks and analysis.
- *Advancement in WASM Runtime Development.* We provide new insights into WASM runtime development and illustrate how SwFuzz can uncover and help fix vulnerabilities, thereby contributing to the improvement and security of WASM runtime environments.
- *Discovery and Patching of Vulnerabilities.* Our proposed SwFuzz not only identifies new vulnerabilities in real-world WASM runtimes but also facilitates the patching of bugs by working collaboratively with developers.

The rest of this paper is organized as follows. Section II introduces the necessary background to understand WebAssembly and fuzzing techniques, and presents our motivation to adopt structure-sensitive WebAssembly fuzzing. Section III gives the high-level idea of the design of our proposed fuzzing framework SwFuzz. Section IV illustrates the implementation details of our SwFuzz. Section V evaluates the bug-finding ability of the proposed fuzzing approach on real-world WASM runtimes, and presents the coverage and ablation studies. Section VI discusses the security of the WASM runtime, and Section VII provides the related work. Finally, we conclude this paper and present the future work in Section VIII.

II. Background and Motivation

A. WebAssembly Format

WebAssembly (WASM) is a binary instruction format designed for stack-based virtual machines, primarily aimed at facilitating the high-speed execution of code on web pages [11]. It serves as a compilation target for source languages such as C, C++, and Rust, allowing developers to run code written in these languages within the browser environment at near-native performance. WASM binaries are encapsulated in a modular format, with each module containing a sequence of sections that provide detailed information such as function definitions, global variables, and data segments. A key feature of the WASM format is its consistent and compact binary encoding, which ensures fast loading times and efficient execution.

More specifically, each WASM binary begins with a 4-byte magic number, followed by a version number that indicates the version of WASM being used [12]. Sections within the binary are identified by unique single-byte code and can appear in any order, subject to certain restrictions. Each section contains details such as type, import, export, function, and memory. During execution, WASM employs a stack-based execution model. Functions within WASM operate on a numeric stack, where operands are pushed onto and popped off the stack. Additionally, WASM's memory model includes a resizable linear memory array and a table of function references, facilitating efficient memory management and function invocation.

B. WebAssembly Runtime

WASM runtimes are specialized environments designed for the execution of WASM modules. These runtimes provide the necessary infrastructure to load, validate, and execute WASM code, thereby ensuring a secure and efficient execution process. WASM runtimes can be broadly categorized into two types, i.e., browser-based runtimes and standalone runtimes.

Browser-based WASM engines [13], [14], such as those integrated into modern web browsers like Chrome, Firefox, and Edge, enable the execution of WASM code alongside JavaScript, providing a seamless integration with existing web technologies. These runtimes take advantage of the security model of web browsers, ensuring that WASM code operates within a sandboxed environment and is isolated from the underlying system. Standalone runtimes, such as Wasmtime [15], WAVM [16], and Wasmer [17], extend the utility of WASM beyond the browser. These runtimes allow WASM modules to run on multiple platforms, including servers, IoT devices, and embedded systems. They provide a lightweight and portable execution environment that can be easily integrated into diverse systems, facilitating the deployment of WASM-based applications across different environments.

C. WebAssembly Workflow

The WebAssembly workflow is a multi-step process that transforms high-level language code into a WASM binary, which can be executed within a browser or other WASM-compliant environment. This process begins with a program written in a high-level language, most commonly C, C++, or Rust. The code is then compiled into a WASM binary, a compact and optimized format designed for fast loading and execution. The WASM binary retains the modular nature of the original code, containing functions, data segments, and other necessary components. Modern web browsers are equipped with built-in support for WASM, allowing them to efficiently load and run these binaries. Upon loading, the WASM binary is instantiated within a sandboxed execution environment. This ensures that it runs safely without posing a threat to the host system. After the WASM module has been executed, it cleans up the resources it has used. Operating in a memory-managed environment, WASM ensures that resources such as memory, table, and other WASM sections are deallocated, promoting optimal system performance and preventing resource leakage.

D. WebAssembly Fuzzing

Fuzzing is a software testing technique that dynamically generates inputs for a program to identify bugs and vulnerabilities. While fuzzing techniques have been widely adopted to test traditional software [18]–[20], newly developed software like WebAssembly presents new challenges to existing fuzzing

methods. WASM binaries are constructed with specialized structures, each of which is parsed during the execution of the WASM file at a given WASM runtime. Based on the parsing result, the code within the WASM file is executed as distinct instructions. This implies that WASM runtimes include both the parsing of structured data and the subsequent code execution based on those parsing results. However, current fuzzing techniques for WASM lack a comprehensive understanding of the structured data. To address this issue, we propose SwFuzz, a structure-sensitive WASM fuzzing framework designed to detect potential vulnerabilities in WASM runtimes.

E. Motivation

Originally designed to improve the performance of web browsers, WASM has demonstrated significant potential across various industries, including blockchain, edge computing, and portable cloud applications [21]. One of the most notable adoptions of WASM has occurred in consortium blockchains, where multiple organizations require a robust platform to execute shared business processes. In this context, WASM's ability to execute code securely and at near-native speeds is particularly valuable. Blockchain platforms such as Substrate [22] and Ethereum 2.0 [23] have adopted WASM as their default execution environment, leveraging its inherent security features and performance advantages.

Deploying WASM in critical systems requires rigorous assurance of reliability and security. Blockchains handle sensitive transactions and data that require high levels of integrity and security. However, the complexity and diversity of WASM bytecode introduce unique vulnerabilities and potential attack surfaces that malicious entities could exploit. Ensuring the robustness of WASM runtimes is therefore paramount. This motivates us to develop an effective vulnerability detection tool for WASM. By employing a structure-sensitive fuzzing approach, we aim to uncover and mitigate vulnerabilities that could otherwise lead to severe security vulnerabilities or system failures. Structure-sensitive fuzzing is particularly well-suited to the binary format of WASM, allowing it to identify deep-seated, complex bugs more effectively than traditional fuzzing methods. This approach focuses on the internal structure of WASM modules, allowing for more precise and thorough testing.

III. Design

In this section, we present the design and methodology of our proposed WASM fuzzing framework. First, we illustrate the overall architecture and workflow of SwFuzz. Then, we introduce the structure-sensitive mutation technology of SwFuzz, including file structure mutation, instruction mutation, and LLMs-assisted mutation.

A. Architecture and Workflow

Figure 1 illustrates the overall architecture and workflow of SwFuzz. When a WASM file is input to the WASM runtime, the process begins by parsing and processing the

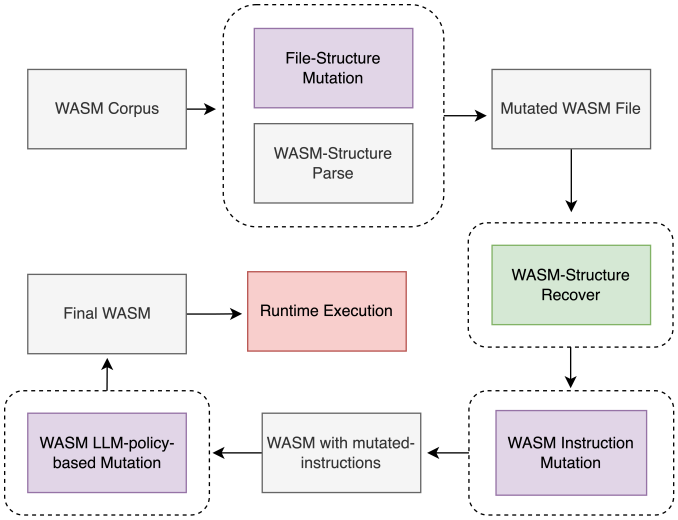

Fig. 1: The overall architecture and workflow of SwFuzz.

structure of the file. The runtime then determines the appropriate actions based on the parsing results. During this initial phase, the WASM runtime initializes the execution state by allocating resources, setting global fields, and performing memory-related operations. Subsequently, the runtime selects and executes the instructions within the WASM file. Each instruction in the WASM virtual machine will be accurately recognized and safely executed, which inherently improves the security of WASM-based applications. SwFuzz's workflow integrates several mutation techniques to adapt to the unique characteristics of the WASM workflow. During the critical stages of file structure parsing, SwFuzz performs structure-sensitive mutations and employs a corresponding structure recovery mechanism to ensure the integrity of the file. This approach tests the robustness of the file structure parsing process. At the instruction parsing and execution stages, SwFuzz applies traditional instruction mutations along with policy-based mutations to improve the overall mutation strategy. To further improve WASM fuzzing, SwFuzz introduces a large language model (LLM)-assisted mutation policy. This advanced policy enhances the instruction mutation and directs the fuzzer to reach deep logic within the WASM runtime, thus increasing the likelihood of uncovering hidden vulnerabilities.

B. File-Structure Fuzzing Stage

Let us first dive into the file structure mutation. Figure 2 shows an overview of our structure-level mutation of WASM files, which can be divided into the following three aspects.

Header Rebuild. Initially, SwFuzz deconstructs the WASM file into its constituent headers and further dissects the binary into distinct sections. This process involves not only parsing the headers but also delving into each section to extract both control information and associated data. This meticulous extraction leverages the native control mechanisms inherent in the WASM format, allowing SwFuzz to accurately map the fundamental structural blueprint of WASM files. By breaking down the WASM binary, SwFuzz can identify and isolate

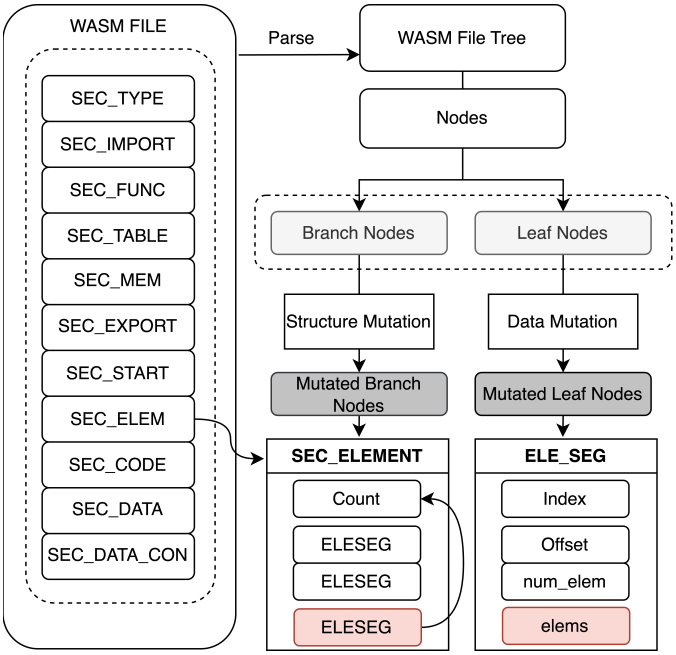

Fig. 2: The structure mutation of WASM files.

key components such as type, import, function, and other sections critical to the execution of WASM modules. Each of these components is carefully analyzed to understand their role and interdependencies within the overall binary structure. This granular level of parsing ensures that SwFuzz has a comprehensive understanding of the layout of the WASM file and the relationships between its various elements.

Mutation. Following the header rebuild, SwFuzz moves on to the mutation phase. By further segmenting the sections into nodes, the framework gains a more detailed understanding and control over the binary, setting the stage for precise byte mutations. SwFuzz categorizes the WebAssembly structure into five distinct fields: *BytesField, RepeatField, UnsignedLeb128Field, SignedLeb128Field,* and *UIntNField*. Based on strategic selection among these fields, SwFuzz implements structure-level mutations, adjusting the subnodes in a manner that introduces variability while preserving the overall structural integrity.

Structure Fix. The structure fix stage is designed to repair the mutated file to a state that is both structurally sound and executable in a WASM runtime environment, without diluting the effectiveness of the introduced mutations. This involves two primary tasks: Length Adjustment and Dependency and Data-Flow Correction.

Length Adjustment. SwFuzz recalculates and adjusts the length attributes of various sections and nodes within the WASM file. Since mutations can change the byte size of sections or elements, adjusting the length fields ensures that the file conforms to the WASM format specifications so that the runtime can correctly parse and execute the file.

Dependency and Data-Flow Correction. Mutations can disrupt the intricate dependencies and data flows between different sections and elements of the WASM module, such as references between function definitions and their calls. The structure fix stage meticulously examines these relationships, repairing broken links and ensuring that data flows and dependencies are logically consistent.

This repair process is essential for maintaining the delicate balance between inducing meaningful mutations to expose vulnerabilities and preserving the executability of the WASM module. Without the structure fixing, there is a significant risk that mutated files would be too corrupted to run, effectively eliminating the potential to discover critical security flaws through dynamic execution and analysis. By implementing this stage, SwFuzz ensures that each mutated file remains a viable candidate for vulnerability detection, thereby maximizing the effectiveness and efficiency of the fuzzing process.

C. Instruction Fuzzing Stage

In the instruction fuzzing stage, SwFuzz transitions to mutating instructions based on the analysis of various fields within the WASM file structure. Upon encountering a *BytesField*, SwFuzz decides whether to use native byte-level mutators or instruction-level mutators. The choice of byte mutators allows SwFuzz to perform mutations that are agnostic to the specific instructions, affecting byte-level data without regard to the instruction semantics. Conversely, selecting instruction mutators prompts SwFuzz to apply mutations that are sensitive to the semantics of the instructions, treating the data at a more granular, instruction-aware level.

Within this stage, SwFuzz employs two primary mutation functions: `insertInstruction` and `eraseInstruction`. The `insertInstruction` function is capable of inserting one of four types of instructions at a random position within the data of a relevant subnode. This allows for the dynamic addition of new behaviors or functionalities within the code structure. The `eraseInstruction` function, on the other hand, facilitates the removal of existing instructions from a subnode, randomly erasing instructions to assess the impact on the program's robustness and behavior. These mutation strategies are designed to explore the robustness of WASM applications by introducing or removing instructions, thereby simulating potential points of failure or vulnerability.

D. LLMs Assisted Fuzzing Stage

In this part, we explore the integration of Large Language Models (LLMs) into the fuzzing process of WebAssembly, specifically focusing on the mutation of WASM instructions once they are converted to WebAssembly text format (WAT) representation. This innovative approach leverages the sophisticated understanding and generation capabilities of LLMs to enhance the complexity and potential vulnerability exposure within the target applications. The mutation process is governed by several distinct policies, each designed to introduce or amplify specific aspects of the code's behavior or structure:

Insertion. Under the insertion policy, LLMs are tasked with inserting new WAT instructions into the existing codebase. This method aims to increase the overall complexity of the application, potentially finding hidden flaws as the

execution paths become more complex. The challenge lies in selecting and positioning new instructions in a way that contributes meaningful complexity without rendering the program non-functional or drastically altering its intended purpose.

Modification. This policy involves that LLMs modify existing portions of the WAT code. The goal is to increase the complexity of the code while ensuring that the control flow is minimally altered. Modifications may include changing variables, operators, or instruction sequences, as long as they do not disrupt the basic logic and flow of the program. Such adjustments require a deep understanding of WAT syntax and semantics, and highlight the role of the LLM in maintaining the delicate balance between innovation and integrity.

Injection. The injection policy focuses on adding potentially vulnerable code snippets to the current WAT code. This strategy aims to simulate real-world attack vectors and test the application's resilience against them. By incorporating known vulnerabilities or patterns that have been exploited in the past, researchers can directly assess the robustness of the application's security measures. The involvement of the LLM ensures that these injections are both contextually relevant and strategically placed, maximizing the likelihood of uncovering significant security issues.

Following the mutation phase facilitated by LLMs, it is critical to perform a semantic check to verify the correctness of the mutated WAT code. This verification process ensures that despite the changes introduced, the mutated code remains valid, executable, and reflects its intended functionality. Semantic checking involves analyzing the mutated code to confirm that there are no syntax errors, that logical structures are preserved, and that there are no unintended changes in the behavior of the program. This step is essential to maintain the balance between exploring the application's security boundaries and ensuring the relevance and applicability of the fuzzing process.

Reduce LLMs Invocation Frequency. Invoking Large Language Models (LLMs) is an expensive operation, both in terms of computational resources and time. Each invocation requires significant processing power and can lead to increased operational costs. To optimize the efficiency of our system and manage resources effectively, we propose a strategy to reduce the frequency of LLM invocations.

Our approach is to invoke large language models (LLMs) only after no new path has been discovered for a period of 5 minutes. This strategy ensures that the LLMs are utilized efficiently, focusing their powerful capabilities on scenarios where conventional methods have plateaued in effectiveness. By doing so, we aim to strike a balance between leveraging the advanced capabilities of LLMs and controlling the associated costs. The benefits of this approach include:

Cost Efficiency. Significant reduction in the cost associated with frequent LLM invocations.

Resource Optimization. More efficient use of computational resources, ensuring that they are allocated where they can have the most impact.

Improved Performance. By invoking LLMs only when necessary, we can maintain high levels of performance without the overhead of constant LLM processing.

Additionally, by delaying the invocation of LLMs, we allow conventional methods to fully demonstrate their effectiveness, thereby further enhancing overall efficiency. This incremental application not only maximizes the utilization of existing resources but also reserves more computational power for critical issues, enabling more precise and efficient solutions.

Runtime	Crash Number	Unique Bug Number
wasm-micro-runtime	41	3
wasm3	53	4
wac	45	7
vmir	415	40

TABLE I: The crash number and the unique bugs number SwFuzz identified in different real-world runtimes.

IV. Implementation

To implement our structure-sensitive WASM fuzzing framework SwFuzz, with related algorithms and components for WASM file structure parsing and WASM instruction execution, we build SwFuzz from scratch. We use AFL as the backbone and implement a separate module using Python to construct the related mutation algorithms and components. We also integrate the LLM-based assistant strategy into SwFuzz using the LangChain framework with ChatGPT-4 [24].

In detail, we hook the main fuzz function in AFL into a crafted Python version which is more suitable for customization and LLMs interaction. Our SwFuzz supports WASM runtimes for different purposes, such as blockchains, browsers, and so on. Moreover, we modularize the SwFuzz and provide different versions of our SwFuzz. SwFuzz-*Format* is used for WASM file parse fuzzing, while SwFuzz-*Instruction* is designed for WASM instruction and file parsing. Due to the modular design and implementation, it can be easily extended to support other fuzzing components in the future.

V. Experiment

In this section, we perform experiments to measure the traceability of our SwFuzz on real-world WASM runtimes. To ensure the effectiveness of the evaluation, we set up the execution environment on a bare-metal machine with an Intel(R) Xeon(R) Platinum 8358 CPU, 15T memory, and 1.0T RAM, running Ubuntu 22.04 LTS version. For each experiment, we limit the runtime to 48 hours. For other fuzzers, we use their default settings and configurations, such as AFL and AFL++.

A. Discovering Zero-Day Vulnerabilities

Table I shows the bug detection results of SwFuzz on four real-world WASM runtimes, including wasm-micro-runtime, wasm3, wac, and vmir. We collect all the crashes detected by SwFuzz. Furthermore, we manually analyze the crashes to check if multiple crashes are attributed to the same root cause. Quantitatively, SwFuzz detects 3, 4, 7, and 40 vulnerabilities on the four WASM runtimes, respectively. Below, we present three case studies to show the effectiveness of SwFuzz in finding new vulnerabilities in real-world WASM runtimes.

```
result = m3_ParseModule (env, &module, wasm,
  ↪  fsize);
if (result) goto on_error;
...
// err module is connected with runtime
result = m3_LoadModule (runtime, module);
if (result) goto on_error;
...
m3_SetModuleName(module, modname_from_fn(fn));
...
// err module trigger err and goto on_error;
result = link_all (module);
    if (result) goto on_error;

    if (wasm_bins_qty < MAX_MODULES) {
        wasm_bins[wasm_bins_qty++] = wasm;
    }
return result;
...
on_error:
    m3_FreeModule(module);
```

Listing 1: Bug logic of Wasm3 Use-After-Free

Case Study 1: Use-After-Free in Wasm3. We reveal several critical bugs in the Wasm3 runtime, which is currently one of the most popular runtimes in the world. During the fuzzing process, we successfully trigger a deep bug by mutating the WASM structure-level data. We show the associated bug logic in Listing 1.

Once our SwFuzz crafts a malicious Wasm binary file, it is first parsed by function `m3_ParseModule`. And when entering the function `m3_LoadModule`, the malicious WASM module is linked to a global-level object, called `runtime`. Due to our crafted module, the `m3_LoadModule` operations will result in an error and fail to load the module. Then break into the error handling logic. However, due to the malicious module having been linked into the global `runtime`, the error handling and finalization operations will result in a use-after-free because of the reference to the error module from global `runtime`.

Case Study 2: Trigger Ignored Logic in Wasm-Micro-Runtime (WAMR). We generate an error module that could be parsed and interpreted maliciously, and we successfully trigger a heap-based-buffer-overflow vulnerability in `*loader_ctx->frame_offset++ = 0;` which the developers previously believe is a benign operation and would not trigger any error. The associated logic is shown in Listing 2.

Case Study 3: Trigger Compatibility of WASI in Wasm-Micro-Runtime (WAMR). WASI (WebAssembly System Interface), known as a critical part of WebAssembly, is often treated as a modular system interface for WebAssembly. As described in the original announcement, it focuses on security and portability. We successfully trigger a WASI-related vulnerability in the corresponding module of WAMR, resulting in a malicious memory address dereference through a crafted read operation. The associated logic is shown in Listing 3.

Until now, we have reported and disclosed all the bugs/vulnerabilities that we found to the corresponding providers. As a result, 7 of them have been fixed under our assistance, and 17 CVEs numbers have been assigned, including CVE-2024-35410, CVE-2024-35418, CVE-2024-35419, CVE-2024-35420, CVE-2024-35421, CVE-2024-35422, CVE-2024-35423, CVE-2024-35424, CVE-2024-35425, CVE-2024-35426, CVE-2024-35427, CVE-2024-25431, CVE-2024-27527, CVE-2024-27528, CVE-2024-27529, CVE-2024-27530, CVE-2024-27532. As for the distribution of different vulnerabilities, we list the results in Table II.

```
if (BLOCK_HAS_PARAM(block_type)) {
    for (i = 0; i <
     ↪  block_type.u.type->param_count; i++) {
#if WASM_ENABLE_FAST_INTERP != 0
        uint32 cell_num =
         ↪  wasm_value_type_cell_num(
            block_type.u.type->types[i]);
        if (i >= available_params) {
            /* If there isn't enough data on
             ↪  stack, push a dummy
             * offset to keep the stack consistent
             ↪  with
             * frame_ref.
             * Since the stack is already in
             ↪  polymorphic state,
             * the opcode will not be executed, so
             ↪  the dummy
             * offset won't cause any error */
            *loader_ctx->frame_offset++ = 0;
            if (cell_num > 1) {
                *loader_ctx->frame_offset++ = 0;
            }
        }
        else {
            loader_ctx->frame_offset += cell_num;
        }
#endif
        PUSH_TYPE(block_type.u.type->types[i]);
    }
}
```

Listing 2: Bug logic of WAMR ignored logic.

```
if (start) {
// Bug trigger
    WASMType *func_type =
        module->functions[start->index -
        ↪  module->import_function_count]
            ->func_type;
    if (func_type->param_count ||
     ↪  func_type->result_count) {
        set_error_buf(error_buf, error_buf_size,
                      "the signature of builtin
                      ↪  _start function is
                      ↪  wrong");
        return false;
    }
}
```

Listing 3: Bug logic of WAMR's WASI check.

More specifically, after conducting a comprehensive study of all the zero-day vulnerabilities we found, we identified the main reason for the difficulty in finding these zero-day vulnerabilities, and why previous security or reliable tests failed to detect them. Specifically, most of the zero-day vulnerabilities we found are in the WASM module handling-related code, which may be related to initialization, error handling, parsing, and so on. However, previous tools lack the ability to test these components in different runtimes, which makes these vulnerabilities hidden in the deep part of the codes. On

Vulnerability Type	Number
Use-After-Free	8
Out-Of-Bound	18
Null-Pointer-Deref	12
Invalid-Memory-Access	10
Denial-of-Service	6

TABLE II: The bug type distribution of vulnerabilities detected by SwFuzz.

the other hand, due to the lack of testing for these parts of the code, developers usually ignore the potential logic errors when trying to handle multiple WASM modules. Furthermore, developers also leave some redundant code in the codebase when performing WASM module parsing operations in the codebase, which eventually leads to the bug trigger. These empirical results show that SwFuzz is able to conduct effective fuzzing in real-world WASM runtimes and detect potential vulnerabilities.

B. Coverage Discussion

To evaluate the effectiveness of SwFuzz in real-world scenarios, we conducted a comprehensive coverage study across different WASM runtimes. Specifically, we targeted the wasm-micro-runtime (WAMR) and wasm3, which are widely used in the industry for deploying WASM applications in various environments. Over a period of 48 hours, SwFuzz was executed on these platforms to collect extensive coverage data, providing insight into the tool's performance and effectiveness.

Our comprehensive coverage analysis demonstrates that SwFuzz significantly outperforms traditional fuzzing tools such as AFL in the context of WebAssembly runtime environments. This enhancement is primarily attributed to SwFuzz's innovative use of structure-sensitive information, which allows for a more nuanced and effective exploration of the codebase. By analyzing the results, we found that the efficiency of SwFuzz outperforms the traditional non-structure-sensitive fuzzers due to the following points:

Ignorant Binary Format Parsing in WASM. Traditional fuzzing tools often struggle with opaque binary formats such as WebAssembly, leading to inefficient and superficial test case generation. SwFuzz overcomes this limitation by implementing an informed parsing mechanism that understands the binary structure of WebAssembly files. This knowledge allows SwFuzz to intelligently navigate and manipulate the binary data, ensuring that the generated inputs are both valid and diverse, leading to more effective and thorough testing.

Mutation and Structure-Level Data Influence. SwFuzz enhances the fuzzing process by integrating structure-aware data mutation strategies. Unlike conventional fuzzers that apply mutations randomly, SwFuzz targets its mutations based on the structural dependencies within the code. This approach ensures that dependent data fields are affected in a manner that respects their contextual relevance, thereby increasing the likelihood of triggering subtle, complex bugs that depend on specific conditions or states.

Combination of Binary Structure Information and Instruction. One of the most innovative aspects of SwFuzz is its ability to combine insights from the binary structure with the executable instructions of the WebAssembly code. This synthesis allows SwFuzz to tailor its fuzzing strategy to the specific characteristics of each instruction, taking into account how it interacts with the binary structure. As a result, SwFuzz can generate more precise and powerful inputs that are capable of exploring deeper code paths and exposing vulnerabilities that are tightly coupled with the architectural and logical design of the application.

C. Ablation Discussion

In this subsection, we aim to understand the different roles within SwFuzz. To provide a comprehensive analysis, we decouple two separate components of SwFuzz, dividing them into SwFuzz-*Structure* and SwFuzz-*Instruction*. To evaluate the effectiveness of these two components, we applied each to multiple targets, including Wasm3, wasm-micro-runtime, vmir, and wac. The results demonstrate that SwFuzz-*Structure* contributes the most to mutation path finding. In comparison, SwFuzz-*Instruction* contributes relatively less to mutation paths. Based on our analysis of the results, we have derived the following insights.

Rethinking the Attacking Surface of WASM Binary Format. Parsing a WASM module is a critical step in the execution of WASM applications. Although vulnerabilities in WASM module parsers are relatively rare, their potential impact is significant. A flawed parser can misinterpret the binary format, resulting in a variety of problems, including Due to the complexity of the WASM binary format, errors can occur in the parsing logic. These errors may go undetected during standard testing but could be exploited by crafted malicious input. Buffer overflows and memory corruption, incorrect handling of input sizes and bounds can lead to buffer overflows, potentially allowing an attacker to execute arbitrary code or cause a denial of service. Type confusion, Incorrect parsing of types and structures in the WASM binary can lead to type confusion where the runtime misinterprets data, resulting in undefined behavior.

Vulnerabilities in the parsing of the WASM module/binary format can have far-reaching consequences for the runtime environment. Exploitation of these vulnerabilities can result in various forms of malicious behavior. (1) Invalid global initialization. A parser vulnerability could allow the creation of invalid global variables or improperly initialized memory, which could be exploited to affect the behavior of the application at runtime. (2) Incorrect module behavior. Flaws in the parsing process can lead to misinterpretation of the module's intended behavior, resulting in security vulnerabilities or unexpected application actions. (3) Malicious memory manipulation. By exploiting parsing vulnerabilities, an attacker could manipulate memory in harmful ways, such as injecting malicious code or altering control flow within the application.

WASM Instruction Attack Surface and Effectiveness. We analyze the effectiveness of fuzzing at the WASM instruc-

tion level by examining the coverage achieved by SwFuzz. Ablation studies show that the number of new execution paths found by fuzzing individual WASM instructions is significantly lower than that found by fuzzing the binary structure of WASM runtimes through structured compilation. This discrepancy can be attributed to several factors.

Simple implementation of lightweight WASM runtimes. Many lightweight WASM runtimes implement instruction execution in a straightforward and minimalist manner. This simplicity reduces the complexity and variability of execution paths, limiting the potential for instruction-level fuzzing to discover new paths.

Limited number of WASM instructions. The WASM instruction set is designed to be compact and efficient, with a limited number of instructions. This limitation inherently limits the range of behaviors and execution paths that can be exercised by instruction-level fuzzing, compared to the more complex and varied binary structures of WASM runtimes.

The results suggest that while instruction-level fuzzing can identify specific vulnerabilities, it is less effective at uncovering new execution paths compared to fuzzing techniques that target the entire binary structure of WASM runtimes. Therefore, a comprehensive fuzzing strategy that includes both instruction-level and binary structure-level fuzzing is recommended to maximize coverage and vulnerability detection in WASM applications.

VI. Discussion: Secure WASM Runtime

WebAssembly (WASM) is becoming increasingly important in a variety of industrial applications. Companies are using WASM to build high-performance, secure, and portable applications. One notable example is the development of private blockchains using WASM as the underlying infrastructure. These blockchains rely on WASM to execute smart contracts and manage transactions securely and efficiently. However, the importance of WASM in these applications means that any vulnerabilities or issues within the WASM environment can result in severe financial and reputational losses. Ensuring the security and robustness of WASM is therefore paramount.

A. Improving the Security Watermark of WASM

Our primary goal is to identify and mitigate vulnerabilities in WASM virtual machines, specifically in the parsing and execution of WASM files. The robustness of a WASM virtual machine has a direct impact on the security of the applications running on it. To achieve this, we use fuzzing techniques to stress test the virtual machine and uncover potential security flaws. Current fuzzing tools, such as AFL (American Fuzzy Lop) and libFuzzer, offer significant power and ease of use, enabling the detection of many issues. However, these tools are generally designed for broader applications and lack specific adaptations for WASM virtual machines. As a result, the level of security achieved with these tools may only be at a generic, industry-wide standard.

To address this gap, we have developed custom fuzzing tools tailored to the unique characteristics of WASM. These specialized tools allow us to uncover vulnerabilities that are otherwise difficult to detect using standard fuzzing techniques. Our approach includes:

Customization of Fuzzing Engines. We customize fuzzing engines to understand the nuances of the WASM binary format and execution model.

Targeted Test Cases. We generate test cases that specifically target known vulnerabilities in the WASM virtual machine, such as boundary checking, memory management, and opcode handling.

Enhanced Coverage Metrics. We implement coverage metrics designed specifically for WASM to ensure that our fuzzing efforts are comprehensive and thorough.

By using these customized tools, we have been able to identify a variety of security issues within WASM virtual machines. This proactive approach significantly reduces the number of undiscovered vulnerabilities, thereby improving the overall security posture of WASM-based systems. The key takeaway is that by reducing the attack surface and proactively addressing potential vulnerabilities, we can ensure a higher level of security for WASM applications.

B. Collects Better Seeds During WASM Runtime Development

Effective fuzzing relies heavily on the quality and diversity of the initial seed corpus. Seeds are the initial input for fuzzing tools, and a well-curated set of seeds can significantly improve the effectiveness of the fuzzing process. Our goal is to collect seeds that provide unique coverage and reveal different facets of the behavior of the WASM virtual machine. We use two primary methods to collect seeds.

Crawling and Minimizing Public WASM Files. We use web crawlers to collect a large number of WASM files from public repositories, focusing primarily on platforms such as GitHub, where many open source WASM projects are hosted. These collected files often exhibit a wide range of programming styles, functionality, and complexity. Using tools such as AFL's cmin script, we distill these samples to create a minimized corpus that retains maximum coverage with the smallest possible set of files.

This approach ensures that our seed corpus contains diverse and representative examples of how WASM is used in the wild, thereby improving the fuzzing tool's ability to detect vulnerabilities.

Collecting Exceptional and Crashing Samples. We also focus on obtaining samples from existing WASM projects that have historically caused virtual machine crashes or exhibited anomalous behavior. These samples are invaluable as they highlight potential edge cases and vulnerabilities. To collect these samples, we manually review issues and vulnerability reports from various WASM projects and repositories. We prioritize high-quality samples that have been linked to security incidents or notable failures. This process often involves close collaboration with the open source community and other stakeholders to ensure we have access to the most relevant and impactful samples.

By integrating these two methods, we create a robust set of initial seeds that include both high-coverage normal samples and critical edge cases. This comprehensive seed corpus enables our fuzzing tools to perform more effectively and identify vulnerabilities that might otherwise go undetected. The end result is a more secure and resilient WASM environment capable of withstanding a wide range of potential threats.

In summary, continuously improving the security and robustness of WASM is essential for its adoption in critical industrial applications. By customizing fuzzing tools and collecting high-quality seeds, we can proactively identify and mitigate vulnerabilities, ensuring that WASM remains a reliable and secure platform for future industry development.

VII. Related Work

A. Fuzzing Technique

Fuzzing techniques have been applied to various domains, showing different approaches and innovations. For example, REDQUEEN [25] serves as a lightweight yet effective alternative to taint tracking and symbolic execution. It optimizes state-of-the-art feedback fuzzing and easily scales to large binary applications and unknown environments. Li et al. [26] propose a program-state-based binary fuzzing approach called Steelix, which increases the penetration power of a fuzzer while maintaining an acceptable slowdown in execution speed. T-Fuzz [27] demonstrates that transforming the target program can lead to more effective bug detection compared to heavyweight program analysis techniques.

Additionally, VUzzer [28] highlights how an application-aware mutation strategy can improve the input generation process of state-of-the-art fuzzers. Qsym [29] introduces a fast concolic execution engine that supports hybrid fuzzing by integrating symbolic emulation with native execution through dynamic binary translation. BandFuzz [30], a novel collaborative fuzzing framework, intelligently coordinates multiple fuzzers using a reinforcement learning algorithm. This algorithm dynamically allocates fuzzing resources to the most effective fuzzer for a given target program, allowing BandFuzz to continuously adapt and refine its fuzzing strategy in response to environmental changes. These advancements in fuzzing techniques illustrate the ongoing evolution and improvement of security testing methods.

B. WebAssembly Fuzzing

One of the most promising approaches towards enhancing WebAssembly security is through fuzzing. For example, Fuzzm [6] is designed as a binary-only fuzzing method for WebAssembly, specifically aimed at detecting and testing memory safety bugs. WasmFuzzer [31] focuses on WebAssembly bytecode level fuzzing. It emphasizes grammar checking within the WASM virtual machine and employs an adaptive mutation strategy to reach deep logic in the WebAssembly environment. Wasm-Semantics-Fuzzer [32] is a WebAssembly implementation through generative fuzzing. This approach involves creating valid WASM programs that self-check their results during execution, helping to detect potential semantic implementation errors. WasmBench [33] provides a large dataset of more than 8,000 real-world WebAssembly binaries, collected from various sources. WARF, the WebAssembly runtime fuzzing project, aims to enhance the security and robustness of WebAssembly VMs, runtimes, and parsers using a variety of fuzzing techniques. Inspired by previous work [34]–[38], we propose the structure-sensitive fuzzing framework SwFuzz, which is built on top of AFL, excepting to facilitate research in the field of WebAssembly fuzzing.

C. WebAssembly Protection

Another effective approach to improving WebAssembly security is to protect WebAssembly from various types of attacks. Several methods and tools have been developed for this purpose. vWasm and rWasm [39] are verified sandboxing compilers for WASM. vWasm utilizes traditional machine-checked proofs to ensure security, while rWasm provides a provably secure sandboxing compiler with competitive runtime performance. eWASM [40] introduces a framework for software fault isolation (SFI) using WASM on resource-constrained embedded systems. This framework enhances the security and robustness of WASM in resource-constrained environments. Sledge [41] facilitates low-latency serverless computing at the edge by leveraging WASM. Lehmann et al. [42] provide a systematic analysis of the binary security of real-world WebAssembly applications. Swivel [43] strengthens WebAssembly against Spectre attacks by ensuring that potentially malicious code cannot exploit these attacks to escape the WASM sandbox. These methods improve the security posture of WebAssembly by addressing various aspects of its execution environment and potential attack vectors. CAMP [44] and ShadownBound [45] utilize compiler-based optimization to enhance user-space memory security which is also compatible with protecting WASM.

VIII. Conclusion and Future Work

In summary, SwFuzz marks an improvement in security testing for WebAssembly, employing structure-sensitive fuzzing to adeptly uncover unique, structurally related bugs in WebAssembly runtimes that traditional tools might miss. Its ability to provide comprehensive code coverage ensures that even obscure code paths prone to vulnerabilities are thoroughly examined. The success of SwFuzz in detecting a wide array of unique bugs emphasizes its importance in environments where security and reliability are paramount, such as consortium blockchains. By integrating SwFuzz into the development lifecycle, organizations can strengthen the resilience and trustworthiness of their WASM applications, paving the way for safer, more reliable software deployment across various industries.

In the future, we plan to extend SwFuzz to support more programming languages that are used in WebAssembly runtimes, beyond the currently supported ones, which could increase its utility. Different languages may introduce unique challenges and vulnerabilities when compiled into WebAssembly, thus broadening the applicability of the tool.

REFERENCES

[1] AWS, "What is web3?" 2023. [Online]. Available: https://aws.amazon.com/what-is/web3/

[2] P. P. Ray, "Web3: A comprehensive review on background, technologies, applications, zero-trust architectures, challenges and future directions," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 213–248, 2023.

[3] S. Zheng, H. Wang, L. Wu, G. Huang, and X. Liu, "Vm matters: A comparison of wasm vms and evms in the performance of blockchain smart contracts," *arXiv preprint arXiv:2012.01032*, 2020.

[4] Y. Yan, T. Tu, L. Zhao, Y. Zhou, and W. Wang, "Understanding the performance of webassembly applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 533–549.

[5] P. P. Ray, "An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions," *Future Internet*, vol. 15, no. 8, p. 275, 2023.

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

[7] B. Jiang, Z. Li, Y. Huang, Z. Zhang, and W. Chan, "Wasmfuzzer: A fuzzer for webassembly virtual machines," in *34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022*. KSI Research Inc., 2022, pp. 537–542.

[8] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *WOOT @ USENIX Security Symposium*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:221178641

[9] A. Fioraldi and D. Zhang, "Libafl: A framework to build modular and reusable fuzzers," *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:251515350

[10] "libfuzzer – a library for coverage-guided fuzz testing." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[11] 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*, 2017, pp. 185–200.

[12] "Binary format." [Online]. Available: https://webassembly.github.io/spec/core/binary/index.html

[13] "spidermonkey." [Online]. Available: https://spidermonkey.dev/

[14] "Javascriptcore on webassembly." [Online]. Available: https://mbbill.github.io/JSC.js/

[15] "A fast and secure runtime for webassembly." [Online]. Available: https://wasmtime.dev/

[16] "Wavm is a webassembly virtual machine, designed for use in non-browser applications." [Online]. Available: https://wavm.github.io/

[17] "Truly universal apps with webassembly." [Online]. Available: https://wasmer.io/

[18] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[19] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2197–2213.

[20] P. Qian, H. Wu, Z. Du, T. Vural, D. Rong, Z. Cao, L. Zhang, Y. Wang, J. Chen, and Q. He, "Mufuzz: Sequence-aware mutation and seed mask guidance for blockchain smart contract fuzzing," *arXiv preprint arXiv:2312.04512*, 2023.

[21] 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*, 2022, pp. 3–8.

[22] "substrate." [Online]. Available: https://substrate.io/

[23] C. Kim, "Ethereum 2.0: How it works and why it matters," *Coindesk: https://www. coindesk. com/wp-content/uploads/2020/07/ETH-2.0-072120. pdf*, 2020.

[24] "Langchain," 2024. [Online]. Available: https://github.com/langchain-ai/langchain

[25] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:85546717

[26] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:39960933

[27] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: Fuzzing by program transformation," 2018, pp. 697–710. [Online]. Available: https://api.semanticscholar.org/CorpusID:4662297

[28] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Network and Distributed System Security Symposium*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:2354736

[29] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym : A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX Security Symposium*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52047247

[30] W. Shi, H. Li, J. Yu, W. Guo, and X. Xing, "Bandfuzz: A practical framework for collaborative fuzzing with reinforcement learning," *ICSE/SFBT 2024*, 2024.

[31] B. Jiang, Z. Li, and Y. Huang, "Wasmfuzzer: A fuzzer for webassembly virtual machines," 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:253513716

[32] "Wasm semantics fuzzer," 2022. [Online]. Available: https://github.com/secure-foundations/wasm-semantics-fuzzer

[33] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:232173900

[34] A. Hazimeh, A. Herrera, and M. Payer, "Magma," vol. 4, no. 3. Association for Computing Machinery (ACM), nov 2020, pp. 1–29. [Online]. Available: https://doi.org/10.1145%2F3428334

[35] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," vol. 47, 2018, pp. 2312–2331. [Online]. Available: https://api.semanticscholar.org/CorpusID:102351047

[36] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. W. Hicks, "Evaluating fuzz testing," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52127357

[37] Y. Chen, D. Mu, J. Xu, Z. Sun, W. Shen, X. Xing, L. Lu, and B. Mao, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 633–645. [Online]. Available: https://doi.org/10.1145/3321705.3329828

[38] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[39] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-safe multilingual software sandboxing using WebAssembly," in *Proceedings of the USENIX Security Symposium*, August 2022.

[40] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova, "ewasm: Practical software fault isolation for reliable embedded devices," vol. 39, 2020, pp. 3492–3505. [Online]. Available: https://api.semanticscholar.org/CorpusID:222186323

[41] P. K. Gadepalli, S. P. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a serverless-first, light-weight wasm runtime for the edge," 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:228085728

[42] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *USENIX Security Symposium*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:219979816

[43] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. M. Tullsen, and D. Stefan, "Swivel: Hardening webassembly against spectre," *ArXiv*, vol. abs/2102.12730, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:232046158

[44] Z. Lin, Z. Yu, Z. Guo, S. Campanoni, P. Dinda, and X. Xing, "CAMP: Compiler and allocator-based heap memory protection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[45] Z. Yu, G. Yang, and X. Xing, "ShadowBound: Efficient memory protection through advanced metadata management and customized compiler optimization," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.