WasmCFuzz: Structure-aware Fuzzing for Wasm Compilers

Xiangwei Zhang
Tianjin University
China

Junjie Wang
Tianjin University
China

Xiaoning Du
Monash University
Australia

Shuang Liu
Tianjin University
China

ABSTRACT

WebAssembly (Wasm) has emerged as a pivotal technology for web applications, offering near-native execution speeds and bolstered security through sandboxed execution. Despite its widespread adoption in major browsers, the rapid evolution of Wasm introduces novel attack surfaces, particularly in Wasm compilers. The challenge of Wasm compiler testing lies in producing semi-valid Wasm samples that are structurally sound enough to bypass initial checks yet sufficiently unique to probe for vulnerabilities. In response, we introduce WasmCFuzz, an innovative fuzzing approach that utilizes AFL-generated random bytes to create semi-valid Wasm formats. This method effectively balances structural validity with the potential to uncover compiler corner cases. Our comprehensive evaluation demonstrates that WasmCFuzz not only outperforms existing methods like Wasm-smith and WAfuzzer but also uncovers 13 previously unidentified bugs in mainstream browsers within just a week. These findings highlight WasmCFuzz's capability in enhancing the security of Wasm compilers, marking a significant step forward in Wasm compiler testing.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Fuzzing, WebAssembly, Browser

ACM Reference Format:
Xiangwei Zhang, Junjie Wang, Xiaoning Du, and Shuang Liu. 2024. WasmCFuzz: Structure-aware Fuzzing for Wasm Compilers. In *2024 ACM/IEEE 4th International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) and 2024 IEEE/ACM Second International Workshop on Software Vulnerability (EnCyCriS/SVM '24), April 15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3643662.3643959

EnCyCriS/SVM '24, April 15, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0565-6/24/04…$15.00
https://doi.org/10.1145/3643662.3643959

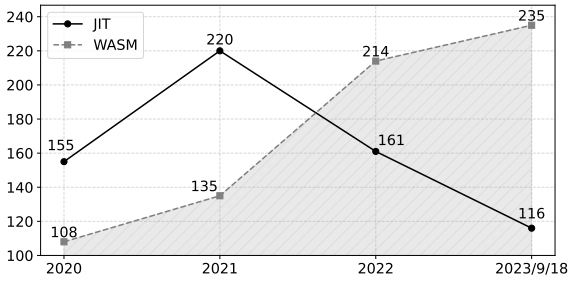


Figure 1: The number of git commits of JavaScriptCore in JIT compiler and Wasm compiler respectively in recent years.

1 INTRODUCTION

WebAssembly (Wasm) is increasingly fundamental to modern web applications, offering a blend of near-native performance and enhanced security through sandboxed execution. As Wasm's popularity grows, with over 92% of browser installations now supporting it [9], its evolving landscape, extending beyond browsers to cloud computing [13], IoT [15], and edge computing [7], presents new security challenges. The rapid evolution of Wasm, particularly with the upcoming Wasm 2.0 specification, introduces a myriad of novel features [14] and, consequently, potential new attack surfaces.

Figure 1 presents a compelling visualization of the increasing focus on Wasm within the realm of JavaScriptCore development, as reflected in the number of Git commits over recent years. Each data point represents the total number of Git commits in a given year, differentiating between those dedicated to the JIT compiler and those focused on the Wasm compiler. Notably, the graph reveals a significant and steady increase in commits related to the Wasm compiler, underscoring the growing importance and complexity of Wasm in modern web browsers. In the realm of web application security, the significance of robust Wasm compilers is underscored by notable incidents where vulnerabilities were exploited. A striking example of this occurred at Pwn2Own 2021 [4], a well-regarded computer hacking contest. During this event, researcher Jack Dates identified and exploited a vulnerability in the Wasm compiler of JavaScriptCore [2]. This exploitation led to remote code execution in Apple's Safari browser [5]. This incident not only demonstrates the real-world implications of vulnerabilities in Wasm compilers but also serves as a wake-up call for the need for more sophisticated security measures in this domain.

Wasm is distinguished by its structured format, vital for ensuring cross-platform execution efficiency. Each Wasm file commences with a magic header signifying its format as Wasm, followed by a

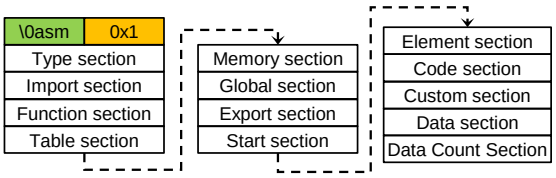


Figure 2: The basic structure of a Wasm binary.

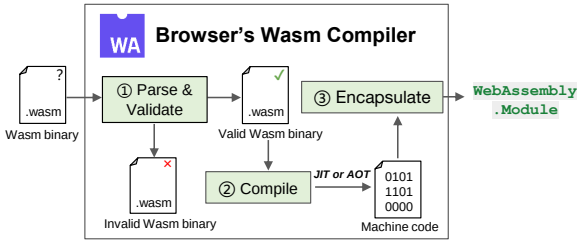


Figure 3: The workflow of browsers' Wasm compiler.

four-byte version number, as depicted in Figure 2. Beyond this, a Wasm file is methodically organized into sections, each identified by a unique ID, ranging from 0 to 12. These sections, necessary for the file's functionality, must follow a specific sequence, reminiscent of how executables are organized in memory. The diversity of these sections caters to different functional aspects of Wasm modules. The Type section specifies function return values and parameter types, while the Import and Export sections manage interactions with external modules. The Function section lists available functions, and the Code section contains their implementations. Additionally, the Table and Memory sections define structures essential for indirect calls and linear memory allocation, respectively. Furthermore, the Global section is where global variables are declared. The Start section identifies an initialization function, and the Data section is responsible for initializing memory segments, as detailed in [6]. Moreover, the Data Count section declares the number of data segments in a module in advance, thereby optimizing the module's loading and compilation processes. The Element section initializes tables with function references, mainly used for indirect function calls. Lastly, the Custom section provides a flexible space for vendor-specific information, useful for debugging or other non-mandatory purposes. This meticulous organization of sections ensures that Wasm modules are not only complete but also optimized for efficient and effective execution across various platforms.

The crux of ensuring Wasm's security lies in the robustness of Wasm compilers. However, the dynamic nature of Wasm's development brings forth the challenge of effectively testing these compilers. The general workflow of a Wasm compiler, as illustrated in Figure 3, begins with the parsing and validation of a Wasm input. Only inputs that comply with the Wasm structure are compiled into machine code, while those that do not adhere to this structure are rejected by the compiler.

Fuzzing, renowned for its effectiveness in detecting bugs in complex software systems, has been adeptly applied to Wasm compiler testing, yielding noteworthy advancements. A significant contribution in this area was presented at Blackhat USA 2022 by Hai

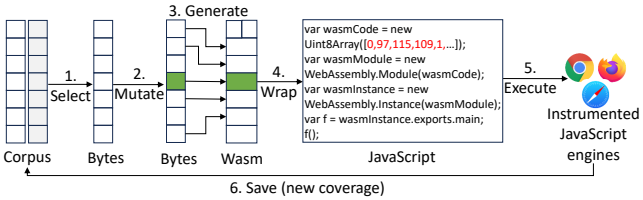


Figure 4: The general workflow of WasmCFuzz.

Zhao and colleagues. They introduced WAfuzzer [7], a coverage-guided Wasm fuzzer developed on the backbone of Libfuzzer [12]. WAfuzzer's innovation lies in its approach to generating Wasm samples: it maps random bytes produced by Libfuzzer into a Wasm format, guided by a set of self-defined rules. This method allows for targeted exploration of the Wasm compiler's potential vulnerabilities. Continuing this trend, Blackhat Asia 2023 witnessed another notable development presented by Zong Cao and the team. Their coverage-guided Wasm fuzzer, based on the AFL (American Fuzzy Lop) [16], employs a similar concept but with a different execution. It generates Wasm samples by transforming AFL's random bytes into Wasm format, utilizing the Wasm generator integrated within the V8 engine [3]. This approach further underscores the versatility and effectiveness of fuzzing in uncovering hidden flaws in Wasm compilers. In contrast to these coverage-guided methods, Wasm-smith [1] offers a different approach as a Wasm test case generator. It operates without coverage guidance, focusing instead on the comprehensive generation of Wasm test cases.

Traditional methods struggle to generate semi-valid Wasm samples that are structurally sound to pass initial validation yet sufficiently varied to probe deeper for vulnerabilities. To address this, we present WasmCFuzz, a novel structure-aware fuzzing approach for Wasm compilers. Our method innovatively leverages AFL-generated random bytes to produce semi-valid Wasm formats, striking a balance between structural integrity and the potential to uncover hidden bugs. Notably, WasmCFuzz demonstrates remarkable efficacy, uncovering 13 previously unknown bugs in mainstream browsers within a week, and outperforming established methods like Wasm-smith and WAfuzzer in both bug detection and code coverage.

2 APPROACH

In developing WasmCFuzz, our overarching objective was to create a fuzzing tool that not only adheres to the complex specifications of Wasm but also introduces a novel approach to uncovering latent vulnerabilities in Wasm compilers. The general workflow of WasmCFuzz, as depicted in Figure 4, is a testament to this dual objective. The process begins with a repository of samples stored in byte format within a corpus. The samples in the initial corpus are stochastically generated, with a random stream of bytes stored in each sample. The first step in our workflow involves selecting a specific sample from this corpus, which will act as the foundation for our mutation operations (Step 1). This chosen sample is then subjected to a variety of mutation techniques, including 1-bit, 2-bit, 4-bit, and 1-byte flips, amongst others (Step 2). Following the

mutation stage, each newly generated sample undergoes a byte-level mapping process to construct a valid Wasm sample (Step 3). Subsequently, these Wasm samples, still in their byte format, are integrated into a JavaScript template (Step 4). This integration is key to preparing the samples for execution within a target JavaScript engine, specifically one that has been instrumented for our testing purposes (Step 5). Conversely, samples that result in new code coverage during execution are earmarked as particularly effective. These samples are then added back into the corpus, ready to be utilized in the next iteration of fuzzing (Step 6).

2.1 Mutator

Key among the mutation strategies is the sequential bit flips. These flips are not uniform but vary in length and stepovers, offering a nuanced approach to altering the binary data. This variability is crucial in exploring a wide range of potential fault lines within the compiler. Additionally, the mutation engine employs operations involving the sequential addition and subtraction of small integers. This operation subtly alters numeric values, probing the compiler's handling of edge cases and boundary conditions. Another critical feature of the mutator's arsenal is the sequential insertion of known interesting integers. This includes strategically chosen values such as 0, 1, INT_MAX (the maximum value for an integer), and others. These particular integers are often instrumental in triggering atypical behavior or revealing oversights in compiler design. These operations are meticulously designed to iteratively challenge and test the compiler, ensuring a thorough and rigorous examination of its capabilities and weaknesses.

2.2 Wasm generator

The Wasm generator, a central component of WasmCFuzz, skillfully utilizes output samples from the mutator as its primary input. This integration is crucial for the generation of structurally valid and yet diverse Wasm files, a balancing act that is at the heart of effective fuzzing. To achieve this, the Wasm generator is meticulously programmed to align closely with the Wasm specification, ensuring every generated file adheres to the required standards. To ensure this structural validity, the Wasm generator operates on a fundamental principle: maintaining every field within its correct value range as defined by the Wasm specification. This principle is achieved by categorizing the fields into four distinct types, each with its unique method of value assignment.

Fields with fixed Values. These fields have predetermined values that are constant and unchanging. For such fields, the Wasm generator directly assigns the correct fixed values as defined in the Wasm specification. This ensures that essential structural elements of the file are consistently accurate.

Fields with calculated Values. Some fields derive their values based on other fields within the Wasm file. For these calculated fields, the generator sets their values dynamically, only after all the dependent fields have been assigned. This sequential setting ensures that the interdependencies among fields are accurately maintained, preserving the logical structure of the file.

Fields with enumerated Values. For fields that are based on a set of predefined options or enumerations, the generator employs a method of selection based on the mutator's output. It reads a number from the mutator and uses this number to select an appropriate value from the available enumeration. This method allows for controlled variability within the confines of the specification.

Fields with randomized Values. Finally, for fields that can accommodate a range of values, the generator utilizes the mutator's output stream to assign randomized values. This approach introduces a degree of unpredictability and diversity in the file, which is crucial for effective fuzzing. It allows the generator to explore a wide array of scenarios within the structural limits of the Wasm format.

2.3 Wrapper

To effectively test the Wasm compiler within JavaScript engines, a crucial step in our WasmCFuzz process involves encapsulating the Wasm file within a JavaScript file. This encapsulation is meticulously carried out to ensure seamless execution and testing. Initially, the Wasm file is encoded as a byte stream and stored in an unsigned 8-bit array, a format conducive for JavaScript processing. This array serves as the foundational element for creating a Wasm module. Once the byte stream is securely embedded within the array, we proceed to instantiate a Wasm module from it. This step is pivotal as it transforms the raw byte data into a functional Wasm module, ready for execution within a JavaScript environment. After the module's creation, the workflow advances to initializing a new instance of this module. This instance is vital as it brings the module to an operational state. From this operational instance, we then export the entry function – the primary function designated to kick-start the Wasm file's execution. The final step in this process is the invocation of the exported entry function. This action triggers the execution of the Wasm file within the JavaScript engine, allowing us to assess the compiler's performance and robustness. By executing the Wasm file in this controlled manner, WasmCFuzz can effectively analyze how the Wasm compiler handles various scenarios and uncover potential vulnerabilities or flaws.

3 EVALUATION

In this section, we present a comprehensive evaluation of WasmCFuzz, showcasing its effectiveness in detecting vulnerabilities in Wasm compilers. Our evaluation leverages the coverage guidance provided by the AFL fuzzing tool, which we have extended to specifically generate Wasm format samples suited for this study.

Testing environment. The entire evaluation was conducted on a high-performance workstation, equipped with an AMD Ryzen 5900X 12-core processor and 32GB of memory, running Ubuntu 20.04.

Testing targets. For a thorough evaluation, we selected three mainstream JavaScript engines that incorporate Wasm compilers: JavaScriptCore, V8, and SpiderMonkey. These engines are integral to major web browsers — Safari, Chrome, and Firefox, respectively.

Baselines for comparison. To contextualize the performance of WasmCFuzz, we compared it against two closely related Wasm fuzzers: WAfuzzer [7] and Wasm-smith [1]. Unfortunately, the complete fuzzer from Paper [3] is not open-sourced, restricting our comparison to just WAfuzzer and Wasm-smith.

Comparison metrics. Our comparative analysis focuses on two key metrics: 1. Bug discovery: we evaluated the number of new bugs

Table 1: Unique bugs detected during 7-days by WasmCFuzz and baselines.

#	ID	Target	Status	WasmCFuzz	Wasm-smith	WAfuzzer
1	1843295	SpiderMonkey	Fixed	✓	x	x
2	1845436	SpiderMonkey	Fixed	✓	x	x
3	1844551	SpiderMonkey	Fixed	✓	x	x
4	258499	JavaScriptCore	Fixed	✓	x	x
5	258795	JavaScriptCore	Fixed	✓	x	x
6	258796	JavaScriptCore	Fixed	✓	x	x
7	258804	JavaScriptCore	Fixed	✓	x	x
8	255805	JavaScriptCore	Fixed	✓	x	x
9	255801	JavaScriptCore	Confirmed	✓	x	x
10	262861	JavaScriptCore	Confirmed	✓	x	x
11	262862	JavaScriptCore	Fixed	✓	x	x
12	262863	JavaScriptCore	Fixed	✓	x	x
13	263363	JavaScriptCore	Confirmed	x	x	✓
14	265927	JavaScriptCore	Fixed	✓	x	x

found in the Wasm compilers over a one-week period. 2. Coverage trend: we also assessed the coverage trend across a single day.

To ensure the reliability of our results and to account for the inherent randomness in experimental outcomes, we repeated each experiment three times. This repetition allows us to present a more accurate and robust analysis of WasmCFuzz's performance in comparison to the baseline tools.

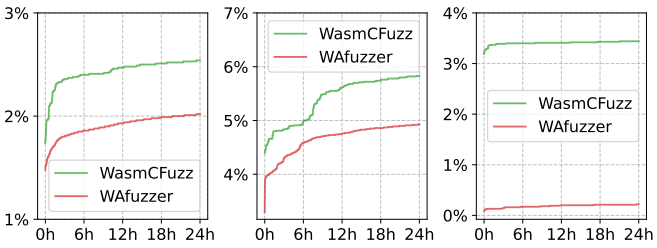
3.1 New bugs found

In the span of just one week, the efficacy of WasmCFuzz was distinctly demonstrated by its ability to identify a significant number of bugs in widely-used Wasm compilers. WasmCFuzz captures those inputs that cause the program to crash. Then, we manually analyze these crash-causing samples and submit a report to the vendor. Specifically, WasmCFuzz detected three bugs in Mozilla's SpiderMonkey engine and an impressive nine bugs in Apple's JavaScriptCore. This achievement is particularly notable considering the mature and extensively tested nature of these compilers. All the bugs identified by WasmCFuzz have been confirmed, and some have already been addressed in the latest versions of the respective engines. For reference, the IDs and details of these bugs can be found in Table 1. All the bugs are accessible in the vendor's bug tracking system by the ID.

When compared to the results achieved by other tools, the superior performance of WasmCFuzz becomes evident. WAfuzzer, for instance, managed to identify only one bug in JavaScriptCore during the same period, while Wasm-smith did not uncover any new bugs.

3.2 Coverage

Beyond the discovery of new bugs, another vital metric for evaluating a fuzzer's effectiveness is its coverage, namely, the extent to which the fuzzer tests various parts of a target's source code. To assess this, we conducted an analysis of the coverage trends over a 24-hour period for both WasmCFuzz and the baselines when applied to Wasm compilers. For this analysis, we focused on comparing WasmCFuzz with WAfuzzer, as Wasm-smith, being a non-coverage-guided tool, did not yield relevant coverage data. The coverage trends observed during the fuzzing of three key Wasm compilers – JavaScriptCore, SpiderMonkey, and V8 – are illustrated in Figure 5.



(a) JavaScriptCore. **(b) SpiderMonkey.** **(c) V8.**

Figure 5: Code coverage trend of WasmCFuzz and the baseline for fuzzing Wasm compilers.

The results after a continuous 24-hour fuzzing session revealed a notable difference in coverage achievements. WasmCFuzz attained 2.54% coverage of the JavaScriptCore source code, surpassing WAfuzzer's coverage, which stood at 2.12%. This represents a significant 19% increase in coverage by WasmCFuzz compared to WAfuzzer. The trend was similar when fuzzing SpiderMonkey and V8, with WasmCFuzz outperforming WAfuzzer by 14% and a remarkable 1583%, respectively. These results indicate that WasmCFuzz not only excels in identifying new bugs but also demonstrates superior performance in terms of code coverage.

4 RELATED WORK

In the landscape of Wasm research, WasmCFuzz is positioned alongside three distinct categories of related work, each contributing uniquely to the field.

Fuzzing Wasm compilers. This category includes tools like Wasm-smith [1], which is adept at generating valid and randomized Wasm modules, albeit without coverage guidance. Another notable work is presented in Paper [11], which develops a stack-directed program generator for differential testing of Wasm compilers in various browsers. WAfuzzer [7] and Paper [3] also target fuzzing Wasm compilers, each employing their own version of coverage guidance. Compared to these, WasmCFuzz distinguishes itself by not only generating more structured Wasm samples but also by integrating advanced features that enhance the depth and efficacy of the fuzzing process.

Fuzzing Wasm binaries. Tools like WAFL [8] propose innovative methods like a lightweight, VM snapshot-based approach for fuzzing Wasm binary code. Fuzzm [10], on the other hand, focuses on inserting stack canaries and mitigating buffer overflows through static binary rewriting. These methods primarily concentrate on securing the Wasm binary itself, rather than the compilers. In contrast, WasmCFuzz zeroes in on the compiler aspect, offering a different but complementary angle of security enhancement.

Wasm attack surface analysis. Lehmann et al [9] delve into identifying attack primitives in Wasm, enabling attackers to manipulate memory, overwrite data, and alter control flow. Such studies underscore the critical importance of tools like WasmCFuzz, as they highlight the potential vulnerabilities in Wasm environments. WasmCFuzz's approach to fuzzing Wasm compilers can be seen as a proactive measure to identify and mitigate such attack vectors before they can be exploited.

5 CONCLUSION

In this research, we introduce WasmCFuzz, a groundbreaking approach to fuzzing Wasm compilers. The cornerstone of WasmCFuzz is its ability to generate format-valid Wasm samples that rigorously adhere to the Wasm specification. WasmCFuzz stands out due to its application of coverage guidance. This technique allows it to intelligently detect new bugs in Wasm compilers by systematically exploring uncharted paths in the code. A testament to the effectiveness of WasmCFuzz is its success in uncovering new bugs in two major mainstream JavaScript engines. These findings are not just a validation of WasmCFuzz's capabilities but also highlight its potential as a valuable tool in enhancing the reliability and security of Wasm compilers. WasmCFuzz is available at https://github.com/WasmCFuzz/WasmCFuzz.

REFERENCES

[1] Bytecode Alliance. wasm-smith. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith.

[2] Apple. JavaScriptCore. https://github.com/WebKit/WebKit.

[3] Zong Cao and Zheng Wang. 2023. Attacking the WebAssembly Compiler of WebKit. In *Blackhat Asia*.

[4] CanSecWest Applied Security Conference. Pwn2Own. https://en.wikipedia.org/wiki/Pwn2Own.

[5] Jack Dates. Exploitation of a JavaScriptCore WebAssembly Vulnerability. https://www.zerodayinitiative.com/blog/2021/4/2/pwn2own-2021-schedule-and-live-results.

[6] C. Gerard Gallant. WebAssembly in Action. https://livebook.manning.com/book/webassembly-in-action/chapter-11/v-8/21.

[7] Zhao Hai, Zhichen Wang, Mengchen Yu, and Lei Li. 2022. Is WebAssembly Really Safe?-Wasm VM Escape and RCE Vulnerabilities Have Been Found in New Way. In *Blackhat USA*.

[8] Keno Haßler and Dominik Maier. 2021. Wafl: Binary-only webassembly fuzzing with fast snapshots. In *Reversing and Offensive-oriented Trends Symposium*. 23–30.

[9] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything old is new again: Binary security of {WebAssembly}. In *29th USENIX Security Symposium (USENIX Security 20)*. 217–234.

[10] Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly. *arXiv preprint arXiv:2110.15433* (2021).

[11] Árpád Perényi and Jan Midtgaard. 2020. Stack-driven program generation of WebAssembly. In *Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30–December 2, 2020, Proceedings 18*. Springer, 209–230.

[12] Kosta Serebryany. 2016. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 157–157.

[13] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers.

[14] WebAssembly. WebAssembly proposals. https://github.com/WebAssembly/proposals.

[15] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 1–4.

[16] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl.