Revealing Performance Issues in Server-side WebAssembly Runtimes via Differential Testing

Shuyao Jiang*, Ruiying Zeng†‡, Zihao Rao†‡, Jiazhen Gu*, Yangfan Zhou†‡, and Michael R. Lyu*

* *Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China*
† *School of Computer Science, Fudan University, Shanghai, China*
‡ *Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China*
syjiang21@cse.cuhk.edu.hk, {ryzeng22, zhrao23}@m.fudan.edu.cn,
jiazhengu@cuhk.edu.hk, zyf@fudan.edu.cn, lyu@cse.cuhk.edu.hk

Abstract—WebAssembly (Wasm) is a bytecode format originally serving as a compilation target for Web applications. It has recently been used increasingly on the server side, *e.g.*, providing a safer, faster, and more portable alternative to Linux containers. With the popularity of server-side Wasm applications, it is essential to study performance issues (*i.e.*, abnormal latency) in Wasm runtimes, as they may cause a significant impact on server-side applications. However, there is still a lack of attention to performance issues in server-side Wasm runtimes. In this paper, we design a novel differential testing approach *WarpDiff* to identify performance issues in server-side Wasm runtimes. The key insight is that in normal cases, the execution time of the same test case on different Wasm runtimes should follow an *oracle ratio*. We identify abnormal cases where the execution time ratio significantly deviates from the *oracle ratio* and subsequently locate the Wasm runtimes that cause the performance issues. We apply *WarpDiff* to test five popular server-side Wasm runtimes using 123 test cases from the LLVM test suite and demonstrate the top 10 abnormal cases we identified. We further conduct an in-depth analysis of these abnormal cases and summarize seven performance issues, all of which have been confirmed by the developers. We hope our work can inspire future investigation on improving Wasm runtime implementation and thus promoting the development of server-side Wasm applications.

Index Terms—WebAssembly, performance issues, differential testing

I. INTRODUCTION

WebAssembly (abbreviated Wasm) is a static low-level bytecode format designed as a portable compilation target for the Web [1]–[3]. Wasm bytecodes are fast to compile and run, portable across browsers and architectures, and provide guarantees of type and memory safety. Such characteristics make Wasm to be increasingly adopted outside the Web context. In particular, Wasm has been considered a better isolation mechanism than containers in cloud environments [4]–[8], since it provides a higher level of abstraction and consumes much fewer resources than typical containers. A state-of-the-art application is *Docker+Wasm* [9], a special build that makes it possible to run Wasm containers with Docker [10] using the WasmEdge runtime [11]. Wasm has also been used in other server-side applications, including microcontrollers [12], [13], trusted execution environments (TEEs) [14] and smart contracts [15]–[17].

With the increase of server-side Wasm applications, studying performance issues of Wasm on the server side becomes highly essential. On the one hand, performance degradation (*e.g.*, latency) in server-side applications usually has a more significant impact than in Web applications. A short latency may not be easily perceived by users in Web applications. But, in some performance-sensitive server applications, it may lead to a decrease in service throughput and cause unexpected economic losses. Our motivating experiment shows that the latency of Wasm runtimes can significantly affect the throughput of some services (the details will be elaborated in Section II). On the other hand, server-side Wasm applications typically run in a standalone runtime system (*e.g.*, WasmEdge [11]). Unlike major browsers (*e.g.*, Chrome, Safari and Firefox) that have been developed for decades and have powerful optimization mechanisms, existing standalone Wasm runtimes are still in the early development stage. Therefore, performance issues of Wasm runtimes are more likely to occur on the server side than on the Web.

However, there is still a lack of research in this area. Existing studies on Wasm performance mainly conducted on the Web environment [18]–[23], while the attention to the server-side Wasm performance is still limited [24]. Moreover, existing research only focuses on the systematic performance gaps between Wasm and native code or JavaScript but lacks attention to *performance issues* in Wasm runtimes. In particular, performance issues refer to the *abnormal latency* occurring in the Wasm runtimes when running specific applications. Such performance issues can usually reveal some inappropriate mechanisms (*e.g.*, code optimization, code execution strategy) of specific Wasm runtimes. Finding performance issues in Wasm runtimes will significantly facilitate the improvement of runtime implementation.

To this end, this paper aims to reveal performance issues in existing standalone Wasm runtimes. However, there are two main challenges to this task. First, there are currently a lot of standalone Wasm runtime implementations (more than 30 Wasm runtimes are held on Github [25]). It is hard to analyze each runtime individually. The second challenge is determining the *oracle* of performance issues, *i.e.*, there is exactly a performance issue in a Wasm runtime. Unlike semantic issues causing failure execution or wrong outputs, there is no ground truth of the performance indicator (*i.e.*, execution time of test cases). Furthermore, a longer execution

time does not directly indicate a performance issue because this may be caused by the features of the test case instead of the runtime implementation.

To address the first challenge, we adopt the idea of differential testing [26]–[29], a typical software testing technique for detecting bugs in a series of comparable systems. The idea is to observe the inconsistency in the outputs of these comparable systems given the same input, which is suitable for testing multiple Wasm runtimes. However, traditional differential testing approaches only target semantic bugs, which cannot be directly applied to performance issues. It is infeasible to identify performance issues simply based on the inconsistency in execution time of the same test case since there are systematic performance gaps among different Wasm runtimes. Therefore, to address the second challenge, we propose a novel differential testing approach *WarpDiff* (***Wa**sm **R**untime **P**erformance **Diff**erential Testing*) for identifying performance issues in Wasm runtimes. The idea is that in normal cases, the execution time of the same test case on different Wasm runtimes should follow a stable ratio, which we call *oracle ratio*. The *oracle ratio* reflects the systematic performance gaps among different Wasm runtimes. Thus, for each test case, we first observe the execution time ratio on different Wasm runtimes, then identify an abnormal case in which this ratio significantly deviates from the *oracle ratio*. For the abnormal case, we further locate which runtime causes the deviation to identify the performance issue.

To evaluate the effectiveness of *WarpDiff*, we apply it to identify performance issues in five popular standalone Wasm runtimes (*i.e.*, Wasmer [30], Wasmtime [31], Wasm3 [32], WasmEdge [11], and WAMR [33]) with different settings. We collect 123 C/C++ programs from the LLVM test suite [34] as our test cases. We compile the test programs to Wasm code by Emscripten [35], then run the Wasm code under each runtime setting and collect their execution time. Based on these data, we identify performance issues in these runtimes by our differential testing approach. We report the top 10 abnormal cases and summarize seven performance issues in four runtimes. We further conduct a comprehensive case analysis of these performance issues to reveal their causes. We report these issues to the developers of the corresponding Wasm runtimes, all of which have been confirmed. Our code and experiment results are all available[1].

The main contributions of this paper are as follows:

- We identify the significance of performance issues in server-side Wasm applications, and we conduct the first study on revealing performance issues in server-side Wasm runtimes.
- We propose a novel and effective differential testing approach *WarpDiff* for identifying performance issues in server-side Wasm runtimes, and we apply it on real-world Wasm runtimes.
- We reveal seven unknown performance issues in four Wasm runtimes and further explain their causes with

comprehensive case analysis. All the issues have been confirmed by the developers.

The rest of this paper is organized as follows. Section II introduces the background of server-side Wasm and illustrates the motivation of our work. Section III describes the design and implementation of *WarpDiff*. Section IV presents our evaluation of *WarpDiff* and analysis of identified performance issues. In Section V, we discuss threats to validity and future work. We describe the related work in Section VI and finally conclude this work in Section VII.

II. BACKGROUND AND MOTIVATION

A. *Wasm on the Server Side*

Wasm is a low-level bytecode language originally intended for client-side execution in the Web [1]–[3]. It serves as the compilation target for applications written in other programming languages such as C/C++, Rust, and Go. Wasm gains popularity in the Web since it is memory-safe, cross-platform, and provides near-native performance [36]. Such attributes also make Wasm to be increasingly used on the server side. In particular, Wasm is a promising solution for running server-side applications in cloud environments [4]–[8]. Compared with the traditional Linux containers, Wasm runtimes are safer since they have fewer attack surfaces. Wasm applications are portable across operating systems and CPU architectures. They can also achieve near-native performance by AOT (ahead-of-time) compilation. Furthermore, Wasm consumes much less memory and fewer resources than Linux containers. In late 2022, Docker [10] announced its support for Wasm with WasmEdge runtime [11] called *Docker+Wasm* [9]. This news means that the application of Wasm on the server side has come into practice.

The operating mechanism of Wasm on the server side is different from that in browsers. To deploy Wasm applications, we first need to compile the source programs written in high-level languages to Wasm bytecode by specific compilers. For example, Emscripten [35] is a popular compiler that compiles C/C++ to Wasm. For Web applications, Emscripten generates Wasm module and JavaScript glue code. During the execution, the JavaScript glue code would call into the browser engine (*e.g.*, V8 in Chrome), which would then talk to the operating system. However, Wasm applications outside browsers need a new way to communicate with the operating system, the WebAssembly System Interface (WASI) [37]. Without a browser engine as runtime, server-side Wasm applications need to run in a standalone runtime system with WASI support. The standalone Wasm runtime works as a sandbox on the host machine, making Wasm applications portable across different platforms. Figure 1 shows the typical workflow of a server-side Wasm application.

With the increase of server-side Wasm applications, many standalone Wasm runtimes have been developed. Currently, more than 30 standalone Wasm runtimes are held on GitHub [25]. One representative runtime is Wasmer [30], which offers exceedingly lightweight containers executable
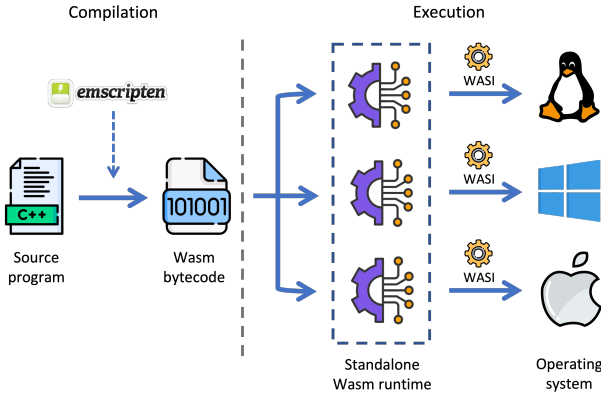
---

[1] https://figshare.com/s/f75ddc64d98669ea3abb

Fig. 1. Typical workflow of a server-side Wasm application.

(a) 10,000 requests

(b) 50,000 requests

Fig. 2. The impact of WasmEdge runtime latency on service throughput under different concurrency and the total number of requests.

from cloud, desktop, or IOT devices. WasmEdge [11] is designed by CNCF [38] and integrated with Docker. Wasmtime [31] and WAMR [33] are two other popular runtimes proposed by Bytecode Alliance [39]. The above runtimes all support AOT compilation. There are also runtimes that execute Wasm code in interpreter mode, such as Wasm3 [32].

However, existing standalone Wasm runtimes are still in the early development stage. Unlike major browsers (*e.g.*, Chrome, Safari and Firefox) developed for several decades, standalone Wasm runtimes are far from mature and more likely to contain issues, especially performance issues. Performance issues are usually harder to reveal during the testing period than semantic issues, but they can have serious adverse impacts on the application.

B. Impact of Performance Issues

High performance is a crucial design criterion of Wasm, and it is one of the attributes that make Wasm popular on both the client side and the server side. However, sometimes there may be performance issues (*i.e.*, abnormal latency) occurring in Wasm runtimes, which is harmful to the reliability of the system. The impact of performance issues in Wasm runtimes on the server side is even more significant than that on the Web. Web applications may not be sensitive to a short runtime latency since they usually have client-side I/O much slower than the runtime latency. On the contrary, server-side applications are usually more performance-sensitive. For example, in some server-side applications with high throughput requirements (*e.g.*, network service), runtime latency may affect the throughput of the application and causes unexpected economic losses.

To study the impact of performance issues in server-side Wasm applications, we conduct a motivating experiment to measure the correlation between Wasm runtime latency and service throughput. Specifically, we select a real-world Wasm microservice [40] with MySQL database backend as our target application. This microservice is a representative server-side Wasm application supported by *Docker+Wasm*, with WasmEdge as the standalone runtime. To simulate the performance degradation in Wasm runtime, we insert a loop
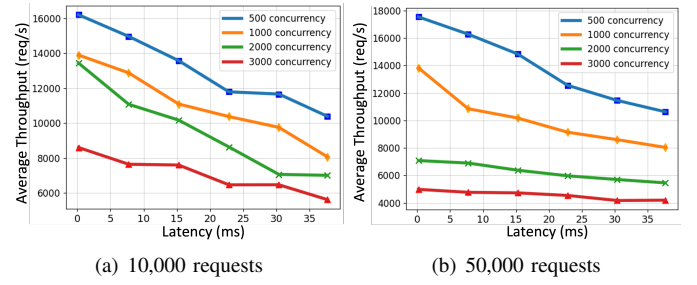
of numerical computing into the request-handling function of the target service. When receiving a request, the service will execute this loop before handling the request. In this way, we can introduce runtime latency without changing the functionality of the target service. We can also control the latency time by changing the number of iterations in the loop. During the experiment, we continuously send requests to the target service from the client machine, and we measure the throughput of the service by *ab* [41], a standard HTTP server benchmarking tool. We fully occupy the CPU during the request handling to ensure the accuracy of the measured throughput.

Figure 2 shows the correlation between the runtime latency of WasmEdge and service throughput under different concurrency and the total number of requests. To eliminate the random error of the measurement, we perform seven replicate experiments for each setting and show the average results. We can find that the runtime latency will cause a severe drop in service throughput under different settings. Specifically, a short latency of 30ms will result in a 20% to 50% drop in service throughput, which is disastrous for high-throughput demanding applications.

Although performance issues can significantly affect the reliability of server-side Wasm applications, there is still a lack of research on performance issues in server-side Wasm runtimes. Existing studies only focus on the systematic performance gaps between Wasm and native code or JavaScript [18]–[24]. To the best of knowledge, none of them has studied performance issues. Therefore, in this work, we aim to reveal performance issues in existing server-side standalone Wasm runtimes and thus facilitate the improvement of Wasm runtime implementation.

III. APPROACH

A. Overview

Finding performance issues in standalone Wasm runtimes is a challenging task. Specifically, there are two main challenges. First, as we have mentioned above, there are many different implementations of standalone Wasm runtimes. It is time-consuming and labor-intensive to analyze each runtime separately. Second, it is hard to determine the *oracle* of performance issues, *i.e.*, how to indicate the occurrence of a performance issue. Unlike semantic issues that usually have
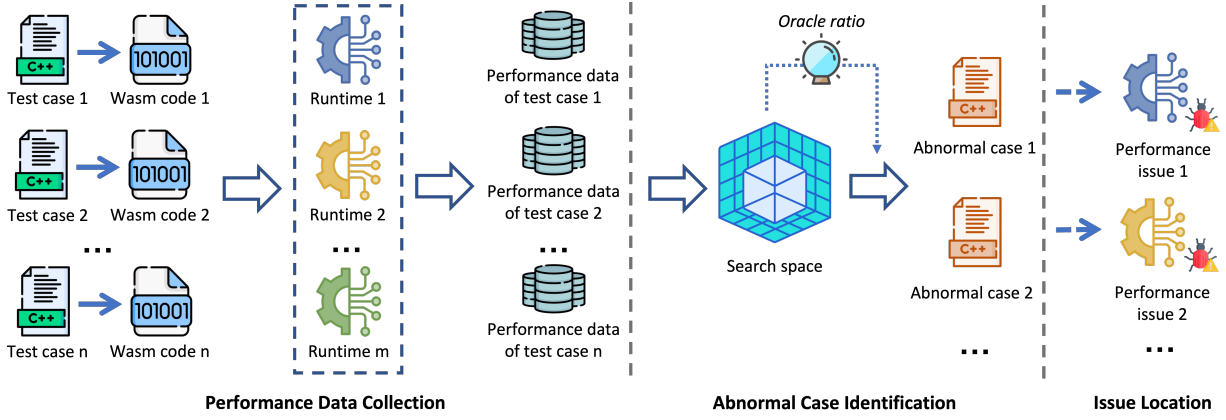
Fig. 3. Overall framework of *WarpDiff* for identifying performance issues in different standalone Wasm runtimes.

a ground truth, performance issues cannot be identified by clear criteria. We cannot identify a performance issue simply by observing the execution time of a test case on a Wasm runtime, because the execution time will be affected by the features of the test case.

Therefore, to address the two challenges, we design a novel and effective approach *WarpDiff* to identify performance issues in different standalone Wasm runtimes. We introduce the idea of differential testing [26]–[29] to solve the first challenge. Differential testing is a widely-used software testing technique for detecting bugs in multiple comparable systems, by providing the same input to these systems and observing the inconsistency in their execution. It is a suitable solution for our task of testing multiple Wasm runtimes. However, existing differential testing approaches only target at semantic bugs, which cannot be directly applied to identify performance issues. The challenge of determining the oracle of performance issues still needs to be resolved. To address this challenge, we introduce a new oracle in *WarpDiff*, which is effective in identifying performance issues in different Wasm runtimes. The key insight is that, in normal cases, the execution time of the same test case on different Wasm runtimes will follow a stable ratio, which we call *oracle ratio*. Although the execution time of a test case on different Wasm runtimes will be affected by the features of this case and the systematic performance gaps among different runtimes, the *oracle ratio* can always be an indicator of normal performance. Therefore, we can identify an abnormal case in which the execution time ratio on different runtimes significantly deviates from the *oracle ratio*. The abnormal cases can indicate performance issues in some Wasm runtimes, and we further locate the specific runtime in which the performance issues occurred.

Figure 3 illustrates the overall framework of *WarpDiff* for identifying performance issues in different standalone Wasm runtimes. Specifically, our approach can be divided into three phases: (1) *Performance data collection*. We execute each test case on multiple Wasm runtimes and collect the performance data; (2) *Abnormal case identification*. We determine the *oracle ratio* based on the performance data of each test case

and identify abnormal cases; (3) *Performance issue location*. We analyze the performance data of the abnormal cases to locate the Wasm runtime with performance issues. In the following subsections, we will elaborate on the details of our design and implementation of these three phases.

B. Performance Data Collection

In order to identify performance issues in different standalone Wasm runtimes, we first need to collect performance data of various test cases on these runtimes. This phase can be further divided into three sub-steps: test case selection, Wasm code execution, and performance data recording.

For test case selection, we need to consider the types of source programs that can be well supported by standalone Wasm runtimes. Currently, Wasm has relatively complete support for source programs written in C/C++ and Rust [36]. Therefore, it is appropriate to select C/C++ or Rust programs as test cases. Furthermore, we should select test cases that are more likely to trigger performance issues in Wasm runtimes, *e.g.*, source programs from some benchmark suites for performance testing.

For each test case, we compile the source program to Wasm code and then execute it on different standalone Wasm runtimes. In this step, we need to ensure the correctness of the execution results of the test cases. We exclude test cases where the execution result is incorrect or a runtime error occurs during execution, because it is meaningless to evaluate the performance of such cases. In order to eliminate random errors of code execution, we execute each test case multiple times on each runtime and record the average value of performance data. The number of executions can be customized according to the requirements for test efficiency.

During the execution of a test case, we need to record the performance data of this test case on each Wasm runtime for differential testing. In this step, we need to consider what performance data to record. The most intuitive idea is that for each test case, record the time of the whole process of its Wasm code running on each runtime. But in order to better locate and analyze the identified performance issues, we record

the performance data with finer granularity. Specifically, the whole running process of Wasm code on a runtime consists of three stages: *runtime initialization*, *Wasm code loading*, and *code execution*. Runtime initialization is where the Wasm runtime starts and prepares the code execution environment. Then in the Wasm code loading stage, the runtimes in AOT mode will first compile the Wasm code to executable binary, while the runtimes in interpreter mode just load the Wasm code into memory. Finally, the runtime performs code execution. Therefore, for each test case, we record the time of these three stages when it runs on each Wasm runtime. For implementation, we use the Linux *perf* tool [42] to find the start and end positions of these three stages and record the time stamps during the test case run. We also record the total time of the whole running process.

C. Abnormal Case Identification

In this phase, we aim to identify abnormal test cases based on the performance data we have collected. For the convenience of data analysis, we only take the *total time* as the performance indicator in this phase (For consistency, we refer to "total time" as "execution time" in the following). The time of the three running stages will be used for further analysis of the identified performance issues.

As we have mentioned above, the key idea of identifying abnormal cases is to observe the execution time ratio on different Wasm runtimes for each case, and the cases where this ratio significantly deviates from the *oracle ratio* are considered abnormal cases. To this end, we need to solve two problems: (1) represent the execution time ratio for each test case; (2) determine the *oracle ratio*.

For the first problem, an appropriate solution is test case vectorization. Specifically, for each test case, we create a time vector to represent the execution time ratio according to the execution time of this case on each Wasm runtime. For example, if the execution time of case x on 3 Wasm runtimes is 1s, 2s, and 3s respectively, the time vector of case x can be represented as $[1, 2, 3]$. However, the time vectors of different test cases cannot be directly compared since the execution time is related to the features of the case itself. Therefore, to make the time vectors of different test cases comparable, we need to normalize the time vectors for all test cases. In this way, the difference in execution time caused by different test cases can be eliminated. Test cases with the same execution time ratio will have the same normalized time vectors. For example, the time vector $[2, 4, 6]$ of case y will be the same as that of case x after normalization.

For the second problem, the ideal solution is that we already know the *oracle ratio*. Unfortunately, the *oracle ratio* cannot be predicted in advance, since the normal performance of each Wasm runtime is currently unknown. The current optimal solution to this problem is to estimate the *oracle ratio* according to the execution time ratio of the existing test cases. Specifically, we have mapped the execution time ratio of all the test cases to the same search space by test case vectorization and normalization. We treat the center (*i.e.*, mean vector) of

all the normalized time vectors as the vector of the estimated *oracle ratio*. Assuming that most test cases are normal cases where the execution time ratio is similar to the *oracle ratio*, when there are enough test cases, our estimated *oracle ratio* will approach the ideal *oracle ratio*.

Thus, for each test case, we can calculate the distance between its normalized time vector and the vector of the estimated *oracle ratio* in the search space. Although the estimated *oracle ratio* will be affected by the abnormal cases, in general, a greater distance means a higher anomaly degree for a test case. Therefore, we rank all the test cases according to this distance, and we identify abnormal cases from the top of this ranking.

D. Performance Issue Location

After we find an abnormal case, we need to locate which Wasm runtime caused this anomaly, *i.e.*, in which runtime a performance issue occurred. To this end, we analyze the impact of each Wasm runtime on this anomaly respectively, based on the execution time of this abnormal case on each runtime. According to our strategy for identifying abnormal cases, the time vector of an abnormal case is relatively far from the vector of the estimated *oracle ratio*. This distance is mainly caused by the abnormal execution time of this case on some Wasm runtimes, *i.e.*, some dimensions with an abnormal value in the time vector. Therefore, we can evaluate the effect of the value of each dimension on this distance separately. Specifically, we adjust the value of one dimension to make the time vector closest to the vector of the estimated *oracle ratio*. We repeat this operation for each dimension and record the value that needs to be adjusted, which we call *deviation degree*. This *deviation degree* reflects the impact of the corresponding Wasm runtime on this abnormal case. The larger *deviation degree* means that this anomaly is more likely to be caused by this runtime. Thus, we treat the Wasm runtime with the largest *deviation degree* as the issue-related runtime.

Hence, for each abnormal case, we can locate the Wasm runtime in which the performance issue occurred by the above solution. It is worth noting that *WarpDiff* is only a heuristic approach, and there may be other solutions for this problem. We just propose a feasible solution and hope our work can inspire more refined approaches in the future. We will show the effectiveness of *WarpDiff* in the next section.

IV. EVALUATION AND ANALYSIS

To evaluate the effectiveness of *WarpDiff*, we apply it on several real-world standalone Wasm runtimes. In this section, we aim to answer the following research questions:

- **RQ1:** How does *WarpDiff* perform in identifying performance issues in real-world standalone Wasm runtimes?
- **RQ2:** What are the causes of the identified performance issues, and how can we verify them?
- **RQ3:** What is the computational overhead of differential testing in *WarpDiff*?

Benchmark	#Program	#LOC*	Benchmark	#Program	#LOC*
Adobe-C++	6	1,615	Misc-C++	7	1,322
BenchmarkGame	8	486	Misc-C++-EH	1	16,817
CoyoteBench	4	1,471	Polybench	30	4,364
Dhrystone	2	642	Shootout	14	573
Linpack	1	693	Shootout-C++	25	783
McGill	4	956	SmallPT	1	96
Misc	27	5,052	Stanford	11	1,135
			Total	141	36,005

* LOC: lines of code.

Runtime	#GitHub Stars*	Test Version	Execution Mode
Wasmer	15.1k	3.2.0	AOT
Wasmtime	12.1k	cli 8.0.0	AOT
Wasm3	6k	v0.5.0	Interpreter
WasmEdge	5.9k	0.12.0	AOT
WAMR	3.7k	1.1.2	Interpreter/AOT

* Statistics of Github stars is by April 2023.

A. Experiment Settings

Test Case Selection. As described in Section III, it is appropriate to select test cases written in source languages that are well supported by Wasm and that are more likely to trigger performance issues. Therefore, we select 141 C/C++ programs with a total of over 30,000 lines of code from the LLVM test suite [34], which contains various benchmark programs for evaluating LLVM compilation performance. We select the test cases from the *SingleSource/Benchmarks/* directory of the test suite, since the programs in this directory can be directly compiled to Wasm code without modification. Table I shows the information of our selected test cases, consisting of 14 benchmarks. One of the benchmarks is Polybench [43], which is a widely-used benchmark suite for Wasm performance evaluation in previous studies [18], [19], [24]. We compile the source programs to Wasm code by Emscripten (version 3.1.24) with the optimization level of *O2*. We exclude those test cases that cannot be compiled successfully or a runtime error occurs during execution. Finally, we collect the results on the remaining 123 test cases.

Wasm Runtime Selection. Although there are many server-side standalone Wasm runtimes, it is better to select some representative Wasm runtimes as test targets. Since most standalone Wasm runtimes are open source on GitHub, we select target runtimes based on their *popularity* and *activity* on Github. For popularity, we select runtimes with the top number of Github stars. For activity, we exclude those unmaintained runtimes, *i.e.*, the last commit was more than one year ago. Finally, we select five representative standalone Wasm runtimes: Wasmer [30], Wasmtime [31], Wasm3 [32], WasmEdge [11], and WebAssembly Micro Runtime (WAMR) [33]. Table II shows the information of these Wasm runtimes. We select the latest version of each runtime for testing. For Wasmer, Wasmtime, and WasmEdge, we test them under AOT mode. Although WasmEdge also provides interpreter mode, the performance is extremely poor, so we only test WasmEdge with AOT mode. For Wasm3, we test it on the default interpreter mode and another setting with *–compile* option, where the lazy optimization of Wasm code will be disabled. For WAMR, we test it on both the interpreter mode and AOT mode. Hence, we finally have seven runtime settings.

Experiment Environment. All our experiments are running on a server with an Intel(R) Core(TM) i5-9500T 2.20GHz CPU and 32GB DDR4 memory. The operating system of the server is 64-bit Ubuntu 20.04.1 SMP with Linux kernel version of 5.15.0-56-generic.

B. RQ1: Results of Identifying Performance Issues

We run each test case 10 times under each runtime setting and collect the performance data averaged over the 10 runs. Then we apply *WarpDiff* on all runtime settings and obtain the results. According to our approach, we identify abnormal cases and then locate performance issues in specific Wasm runtimes based on their *deviation degree* in each case. The larger *deviation degree* indicates that the case performance on the corresponding runtime is with the higher anomaly. Therefore, we rank the identified abnormal cases based on the descending order of the *deviation degree* of the issue-related runtime. Table III shows the results of the top 10 abnormal cases. We only report the top 10 abnormal cases since we just rank the cases without setting a specific threshold for abnormal cases. We design this strategy because our goal is to reveal some unknown performance issues in existing standalone Wasm runtimes, instead of finding all the performance issues. Actually, it is impossible to find all the performance issues, because there is currently no ground truth of performance issues that can be verified.

The values in Table III represent the *deviation degree* of each Wasm runtime setting on the top 10 abnormal cases. A positive value means that the execution time of this case on this Wasm runtime is higher than the expected value according to the *oracle ratio*, while a negative value means that the execution time is lower than the expected value. Since we aim to identify performance issues (*i.e.*, the execution time is abnormally higher than expected), we only focus on the *deviation degree* with positive values. For each abnormal case, the issue-related runtime is marked with a gray background in the table. We can observe that among the 10 abnormal cases, four cases are caused by WAMR with interpreter mode, and the other six cases are caused by Wasmer, Wasmtime and WasmEdge (two cases on each runtime). There are also other abnormal cases caused by Wasm3, which are not shown in the table.

The results indicate that performance issues are common in existing popular standalone Wasm runtimes, which need our attention. We will further conduct a detailed case analysis to reveal the causes of these performance issues.

Case	Wasmer	Wasmtime	Wasm3	Wasm3_compile	WasmEdge	WAMR	WAMR_AOT
BenchmarkGame/fasta.c	0.702	0.113	-0.248	-0.244	0.082	-0.270	0.081
Shootout/methcall.c	-0.051	-0.028	-0.164	-0.164	0.502	0.044	-0.014
Shootout-C++/methcall.cpp	-0.036	-0.031	-0.126	-0.128	0.415	0.072	-0.009
Shootout/random.c	0.075	0.315	-0.060	-0.060	0.079	-0.026	0.101
Shootout-C++/random.cpp	0.096	0.309	-0.063	-0.063	0.098	-0.036	0.121
Polybench/2mm.c	-0.038	-0.039	-0.151	-0.149	-0.035	0.268	0.003
Polybench/gemm.c	-0.038	-0.041	-0.145	-0.153	-0.036	0.267	0.007
Polybench/3mm.c	-0.037	-0.040	-0.145	-0.140	-0.034	0.261	0.005
Misc/flops-8.c	-0.019	0.012	-0.142	-0.142	-0.009	0.251	0.015
Misc/flops-4.c	0.234	-0.003	-0.127	-0.127	-0.019	0.168	0.001

C. RQ2: Case Analysis

In order to verify the identified performance issues and further facilitate the improvement of Wasm runtime implementation, it is critical to analyze the causes of these performance issues. Unfortunately, since the performance issues we identified are all unknown issues, there are no ground truths that can be directly used for verification. Therefore, we conduct a manual analysis of these abnormal cases. Specifically, we analyze each case in three steps: *abnormal stage location*, *fine-grained cause location*, and *cause verification*.

In the first step, we locate the running stage where the abnormal latency occurs. As mentioned in Section III, we have collected the time of the three running stages (runtime initialization, Wasm code loading, and code execution) for each test case. Thus, we can locate the abnormal stage based on these performance data. Similarly, we apply *WarpDiff* on the data of these three stages respectively and identify the abnormal stage where the issue-related runtime (*e.g.*, for case fasta.c, the issue-related runtime is Wasmer) holds the largest *deviation degree*. We find that in all 10 abnormal cases, the abnormal latency occurs in the code execution stage. This means that the performance issues we identified are all caused by the code execution mechanism of the corresponding runtimes.

This finding indicates that we can locate fine-grained causes of the performance issues by analyzing the source code of the abnormal cases. Therefore, in the second step, we aim to find out which part of the code is executing with an abnormal latency. To this end, for each abnormal case, we make a series of case reduction, and we rerun the reduced cases on all the Wasm runtimes to observe the changes in the execution time ratio. Specifically, we generate a reduced case by deleting a code snippet (*e.g.*, a statement, a loop, or a branch). If the execution time ratio of the reduced case changes to the normal level (*i.e.*, close to the *oracle ratio*), it means that the performance issue is likely to be caused by the deleted code snippet, which we call *issue-related code snippet*.

To verify the causes of the performance issues, we further create some new test cases that contain the same function as the *issue-related code snippet*. We run the new test cases on

```
47  static void repeat_fasta (char const *s, size_t count) {
48      size_t pos = 0;
49      size_t len = strlen (s);
50      char *s2 = malloc (len + WIDTH);
51      memcpy (s2, s, len);
52      memcpy (s2 + len, s, WIDTH);
53      do {
54          size_t line = MIN(WIDTH, count);
55          fwrite (s2 + pos,1,line,stdout);
56          putchar ('\n');
57          pos += line;
58          if (pos >= len) pos -= len;
59          count -= line;
60      } while (count);
61      free (s2);
62  }
```

(a) Issue-related code snippet of fasta.c.

```
1   #include <stdio.h>
2
3   static void repeat(int count) {
4       int len = 50;
5       do {
6           count -= len;   // decrease the value of count
7           printf("%d\n", count);  // output the value of count
8       } while (count >= 0);
9   }
10
11  int main() {
12      repeat(500000);  // invoke repeat() with parameter 500000
13      return 0;
14  }
```

(b) A new test case that can reproduce Issue #3784.

Fig. 4. Test cases related to Issue #3784 of Wasmer.

all the Wasm runtimes and observe whether the performance issues will be reproduced. If the performance issues can be reproduced, it means that the causes we found can be verified. Finally, we report the performance issues and their causes to the developers of the corresponding Wasm runtimes.

Overall, we summarize 7 performance issues for the 10 abnormal cases, all of which have been confirmed by the developers. Table IV shows the summary of these performance issues. Next, we will explain the performance issues of each Wasm runtime separately.

Wasmer. In Issue #3784, we find that the core function in the abnormal case fasta.c is repeat_fasta, as shown in Figure 4(a). This function prints the characters of the string s repeatedly, and it stops when the total number of characters

TABLE IV
SUMMARY OF PERFORMANCE ISSUES RELATED TO THE 10 ABNORMAL CASES.

Case	Related Runtime	Issue ID	Cause of Performance Issue	Status
BenchmarkGame/fasta.c	Wasmer	#3784	Improper implementation of fd_write	Confirmed
Misc/flops-4.c	Wasmer	#3821	Version issue of the *Cranelift* code generator	Confirmed
Shootout/methcall.c	WasmEdge	#2444	Improper handling when invoking function pointer	Confirmed
Shootout-C++/methcall.cpp	WasmEdge	#2442	Improper handling of virtual function	Confirmed
Shootout/random.c	Wasmtime	#6287	Insufficient optimization for division and modulo	Confirmed
Shootout-C++/random.cpp	Wasmtime			
Polybench/2mm.c	WAMR	#2175	Insufficient optimization for matrix multiplications	Confirmed
Polybench/gemm.c	WAMR			
Polybench/3mm.c	WAMR			
Misc/flops-8.c	WAMR	#2167	Insufficient optimization for complex arithmetic expressions	Confirmed

printed is *count*. We further locate the *issue-related code snippet*, which accounts for the majority of the case execution time at lines 55-56 by case reduction. Here the program invokes two C standard I/O functions fwrite and putchar in a loop. When we delete these two lines of code, the execution time of this case on Wasmer will go back to normal. Therefore, this performance issue is probably caused by improper I/O implementation of Wasmer. To verify this cause, we create a new test case that also includes a standard I/O function printf in a loop, as shown in Figure 4(b). We find that the performance issue of Wasmer can be reproduced in this case. Then, we check the source code of I/O implementation in Wasmer (in *wasmer/lib/wasi/src/syscalls/wasi/fd_write.rs*), delete the code snippet of setting written size and rebuild Wasmer. We find that the performance issue will not occur after rebuilding. Therefore, the cause of improper I/O implementation in Wasmer can be verified.

In Issue #3821, the abnormal case flops-4.c is a program that calculates the integral of $cos(x)$ using the trapezoidal method. We find that the *issue-related code snippet* of this case is a statement that performs arithmetic operations. Thus, the performance issue may be related to Wasmer's improper handling of such operations. Specifically, Wasm runtimes in AOT mode will generate executable machine code for the input Wasm code before execution. The default code generator of Wasmer is *Cranelift* [44]. When we change the code generator to the LLVM backend and rerun this case on Wasmer, the performance is back to normal. However, Wasmtime also uses *Cranelift* as the default code generator but no performance issue occurs, which indicates that the issue is caused by the current version of *Cranelift* in Wasmer.

These two performance issues of Wasmer have been confirmed by the developers and marked as milestones for the development of the next version.

WasmEdge. In Issue #2444, the abnormal case methcall.c defines a structure named *Toggle*, and it invokes a function to activate the toggle repeatedly. For the convenience of explaining the issue, we create a simplified methcall.c, as shown in Figure 5. In this case, we locate the *issue-related code snippet* at line 20, where the program

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct Toggle {  // define a structure of Toggle
5      char state;
6      void (*activate)(struct Toggle);
7  } Toggle;
8
9  void toggle_activate(Toggle this) {  // activate the toggle
10     this.state = !this.state;
11 }
12
13 int main() {
14     int i, n = 1000000;
15     Toggle tog;
16     tog.state = 1;
17     tog.activate = toggle_activate;
18
19     for (i=0; i<n; i++) {
20         tog.activate(tog);  // invoke the function by pointer
21         // toggle_activate(tog);  // invoke the function directly
22     }
23     puts(tog.state ? "true\n" : "false\n");
24     return 0;
25 }
```

Fig. 5. Simplified methcall.c related to Issue #2444 of WasmEdge.

invokes the function toggle_activate via the function pointer activate defined in the structure $Toggle$. However, when we remove the code of this line and invoke the function toggle_activate directly (as shown in line 21), the performance issue of WasmEdge will not show up again. The results indicate that this performance issue is caused by the improper handling of WasmEdge when invoking a function pointer.

The abnormal case methcall.cpp in Issue #2442 implements the same function as methcall.c, but written in C++. Due to differences in syntax of C and C++, the function pointer activate in methcall.c is defined as a virtual function reference virtual bool& activate() in methcall.cpp. The performance issue in this case also occurs when invoking activate. Therefore, we find that WasmEdge also has improper handling of a virtual function. These two performance issues are also confirmed by the developers of WasmEdge.

Wasmtime. The abnormal cases random.c and random.cpp reveal the same performance issue #6287 of Wasmtime. The core functions of the two programs are generating a random number by some compound operations,

```
16 inline double gen_random(double max) {  // generate a random number
17   static long last = 42;
18
19   last = (last * IA + IC) % IM; // compound operations of *, + and %
20   return( max * last / IM ); // compound operations of * and /
21 }
```

(a) Issue-related code snippet of `random.c`.

```
1  #include <stdio.h>
2
3  int main() {
4    int N = 10000000, last = 42;
5    while (N--) {
6       last = (last + 33) % 13;   // compound operations of + and %
7    }
8    printf("%d\n", last);
9    return(0);
10 }
```

(b) A new test case that can reproduce Issue #6287.

Fig. 6. Test cases related to Issue #6287 of Wasmtime.

```
90  #pragma scop
91    /* D := alpha*A*B*C + beta*D */
92    for (i = 0; i < _PB_NI; i++)
93      for (j = 0; j < _PB_NJ; j++)
94      {
95        tmp[i][j] = 0;
96        for (k = 0; k < _PB_NK; ++k)
97          tmp[i][j] += alpha * A[i][k] * B[k][j];   // alpha*A*B
98      }
99    for (i = 0; i < _PB_NI; i++)
100     for (j = 0; j < _PB_NL; j++)
101     {
102       D[i][j] *= beta;                             // beta*D
103       for (k = 0; k < _PB_NJ; ++k)
104         D[i][j] += tmp[i][k] * C[k][j];            // alpha*A*B*C + beta*D
105     }
106 #pragma endscop
```

Fig. 7. Issue-related code snippet of `2mm.c` in Issue #2175 of WAMR.

as shown in Figure 6(a). We locate the *issue-related code snippet* at lines 19-20, which means that Wasmtime may handle such compound operations improperly. We then create another test case with a similar function, as shown in Figure 6(b), and find that the performance issue will be reproduced. We further create more test cases that contain different compound operations, and we find that this performance issue of Wasmtime only occurs when division and modulo are included. We report this performance issue to the developers of Wasmtime. They confirm this issue and admit that the optimization of division and modulo is currently not well supported by Wasmtime.

WAMR. The three abnormal cases `2mm.c`, `gemm.c`, and `3mm.c` from Polybench reflect the same performance issue #2175 of WAMR in interpreter mode. The functions of these three programs are all matrix multiplication. Figure 7 shows the core computations in `2mm.c`, where the program performs the operation $alpha*A*B*C+beta*D$ on matrices A, B, C, D. We locate the *issue-related code snippet* at lines 97, 102, and 104, which are the statements of matrix multiplication. We also obtain similar results on `gemm.c` and `3mm.c`. In particular, the execution time of these cases on WAMR is more than $2\times$ slower than that in Wasm3 (another Wasm interpreter), while WAMR can achieve comparable performance to Wasm3 on other normal cases. This indicates that WAMR may not optimize the matrix multiplication operation well enough in interpreter mode.

In Issue #2167, the abnormal case `flops-8.c` calculates

```
241 x = piref / ( three * (double)m );        /*********************/
242 s = 0.0;                                   /*   Loop 9.        */
243 v = 0.0;                                   /*********************/
244
245 for( i = 1 ; i <= m-1 ; i++ )
246 {
247   u = (double)i * x;
248   w = u * u;
249   v = w*(w*(w*(w*(w*(B6*w+B5)+B4)+B3)+B2)+B1)+one;
250   s = s + v*v*u*((((((A6*w+A5)*w+A4)*w+A3)*w+A2)*w+A1)*w+one);
251 }
```

Fig. 8. Issue-related code snippet of `flops-8.c` in Issue #2167 of WAMR.

TABLE V
COMPUTATIONAL OVERHEAD OF DIFFERENTIAL TESTING UNDER
DIFFERENT NUMBERS OF RUNTIME SETTINGS.

#Runtime	2	3	4	5	6	7
Avg. Overhead (s)	0.330	0.476	0.604	0.735	0.845	0.966
Std. Deviation	0.026	0.039	0.047	0.058	0.044	0.037

integral of $sin(x) * cos(x) * cos(x)$ from 0 to $PI/3$. The *issue-related code snippet* is shown in Figure 8. We find that the abnormal latency of WAMR occurs when handling complex arithmetic expressions in a loop, like the code at lines 249-250. We also observe this phenomenon in some other similar programs of `flops-8.c`. Therefore, WAMR in interpreter mode may also not have sufficient runtime optimization for such complex arithmetic expressions. We have received confirmation for these two issues.

D. RQ3: Computational Overhead

The efficiency of *WarpDiff* is important for its usability in practice. Therefore, we also evaluate the computational overhead of differential testing in *WarpDiff*. Specifically, we measure the running time of the differential testing part (abnormal case identification and performance issue location) in *WarpDiff*, with different numbers of runtime settings. We exclude the time of performance data collection because this part of the time is determined by test case execution and should not be counted in the overhead of differential testing. For each number of runtime settings, we perform differential testing on all possible runtime setting combinations 10 times and calculate the average running time.

The results are shown in Table V. We can find that as the number of runtime settings grows, the computational overhead of differential testing increases steadily, but all within one second. In our experiments, the time spent on performance data collection for a single execution of all the test cases is about two hours. It means that the computational overhead of differential testing only accounts for less than 0.01% of the whole process. The results indicate that *WarpDiff* is highly efficient, which provides good usability for its practice.

V. DISCUSSION

A. Threats to Validity

There are some threats to validity of our work, including test case selection, Wasm runtime selection, and the sufficiency of case analysis.

First, we select 123 C/C++ programs from the LLVM test suite as our test cases, which may not be very large-scale. However, the test cases are representative benchmark programs for performance testing and are well-suited as the source programs of Wasm. Our test cases include Polybench [43], a popular benchmark that is widely used for Wasm performance evaluation in previous studies [18], [19], [24]. Furthermore, the goal of our work is to reveal some unknown performance issues in server-side Wasm runtimes instead of finding all the performance issues (actually, it is impossible). Our evaluation has shown the effectiveness of the selected test cases in achieving our goal.

Second, we select five server-side standalone Wasm runtimes as our test targets. We select the Wasm runtimes according to their popularity and activity, thereby ensuring the representativeness of the selected runtimes. Also, the number of runtime settings may affect the testing results, as the abnormal cases are identified based on the execution time ratio on these runtime settings. We conduct a series of experiments with different numbers of runtime settings, and we find that those abnormal cases with high *deviation degree* on the issue-related runtime can always be identified.

Third, we only report the top 10 abnormal cases in this paper, as it is inappropriate to set a threshold for abnormal case identification. We report the 10 abnormal cases since they are with the top anomaly degree and worthy of attention. We conduct an in-depth case analysis to reveal the causes of the performance issues. We also report these issues to the developers of the corresponding Wasm runtimes to get their confirmation. The results indicate the effectiveness of our differential testing approach.

B. Future Work

In this work, we propose a novel differential testing approach to identify performance issues in server-side standalone Wasm runtimes. Based on our approach, we can collect more performance issues in existing popular standalone Wasm runtimes, then build a comprehensive benchmark suite for testing the performance issues in Wasm runtimes. This benchmark suite can facilitate future work on performance issue testing for Wasm runtimes.

Also, our work can facilitate the improvement of existing standalone Wasm runtime implementation. In the future, we aim to further improve the internal mechanisms related to the performance issues in existing Wasm runtimes. We can also design a new Wasm runtime implementation with a better optimization strategy and execution mechanism.

VI. RELATED WORK

Server-side Wasm. WebAssembly (Wasm) is a low-level bytecode language originally designed for client-side execution in Web browsers [1]. Wasm's sandboxing execution mechanism brings safety, higher-performance, lightweight, and portability natures, making it suitable for server-side applications as well [45]–[47]. Cloud applications built with Wasm have become increasingly popular in recent years [4]–[8]. For example, FASSM [4] introduces a new isolation abstraction based on Wasm for high performance serverless computing. Wasm is also suggested to enable computational offloading in cloud environments [48]–[51]. Nomad [49] provides a cross-platform computational offloading and migration mechanism in Femtoclouds using Wasm. WIPROG [50] proposes an edge-centric approach to IoT application programming based on Wasm. Besides cloud environments, Wasm is also used in microcontrollers [12], [13], Trusted Execution Environments (TEEs) [14] and smart contracts [15]–[17].

Wasm Performance. High performance is an important design consideration of Wasm. Wasm attempts to provide near-native execution speed both in browsers and server-side applications [1], [36]. Extensive work studies Wasm performance over the browsers [2], [18]–[23]. Jangda *et al.* [18], [19] build BROWSIX-Wasm to run unmodified Wasm-compiled Unix applications directly inside the browser. Then they use BROWSIX-Wasm to conduct the first large-scale evaluation of the performance of Wasm in comparison with native code. They point out a substantial performance gap between the two. Wang [20] investigates how Chrome optimizes Wasm execution in comparison to JavaScript. Yan *et al.* [21] extend this study to more browser engines (Chrome, Firefox, and Edge). They find that JIT optimization significantly impacts JavaScript speed but has little effect on Wasm speed. Also, Wasm uses much more memory than JavaScript. Regarding the server-side Wasm performance, there are relatively fewer studies. Spies *et al.* [24] conduct an evaluation of Wasm performance in non-Web environments. The evaluation demonstrates that Wasm is generally faster than JavaScript and can approach native code performance in some cases. To sum up, existing studies simply compare the performance of Wasm with other codes. There is still a lack of research on how to test performance issues in Wasm runtimes.

Differential Testing. Differential testing is a popular software testing technique for detecting bugs in two or more comparable systems or different implementations of the same application [26]–[29]. The idea is to provide the same input to these comparable systems, and observe the inconsistency in their execution. If the results differ, it indicates that some of the systems may contain a bug. Differential testing has been widely used to detect semantic bugs in diverse domains, such as C compilers [52]–[55], JVM implementations [56]–[59], SSL/TLS implementations [60]–[62], and even deep learning systems [63]–[65]. Existing differential testing approaches can be divided into two categories, unguided and guided, based on the way of input generation. Unguided differential testing approaches generate test inputs independently without considering information from past inputs. An example is Frankencerts [60], which tests for semantic violations of SSL/TLS certificate validation across multiple implementations. Guided differential testing approaches aim to minimize the number of inputs by considering program behavior information for past inputs, making the testing process more efficient. For example, classfuzz [56] is a coverage-guided fuzzing approach for differential testing of JVMs' startup

processes. Existing differential testing approaches only focus on semantic or logic bugs in software systems. In this work, we first extend differential testing to performance issue detection, which is one of our contributions.

VII. Conclusion

Performance issues are critical for server-side Wasm applications, but research in this area is lacking. In this work, we conduct the first study on performance issues in server-side standalone Wasm runtimes. We propose a novel differential testing approach *WarpDiff* to identify performance issues in standalone Wasm runtimes, and we apply it to five popular real-world Wasm runtimes with 123 test cases. We further conduct a comprehensive case analysis of the top 10 identified abnormal cases, and summarize seven performance issues in four popular Wasm runtimes. All issues are confirmed by developers. The results indicate the effectiveness of *WarpDiff*, which provide inspiration for future work on improving server-side Wasm runtime implementation.

Acknowledgment

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," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[2] M. Reiser and L. Bläser, "Accelerate javascript applications by cross-compiling to webassembly," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, 2017, pp. 10–17.

[3] L. Wagner, "A webassembly milestone: Experimental support in multiple browsers," *Mozilla Hacks (14 March 2016).*, 2017.

[4] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 419–433.

[5] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 140–149.

[6] F. Eriksson and S. Grunditz, "Containerizing webassembly: Considering webassembly containers on iot devices as edge solution," 2021.

[7] S. M. Jain and S. M. Jain, "Extending istio with webassembly," *WebAssembly for Cloud: A Basic Guide for Wasm-Based Cloud Apps*, pp. 151–160, 2022.

[8] J. Long, H.-Y. Tai, S.-T. Hsieh, and M. J. Yuan, "A lightweight design for serverless function as a service," *IEEE Software*, vol. 38, no. 1, pp. 75–80, 2020.

[9] *Announcing Docker+Wasm Technical Preview*, https://www.docker.com/blog/announcing-dockerwasm-technical-preview/.

[10] I. Docker, "Docker," *lınea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[11] *WasmEdge*, https://github.com/WasmEdge/WasmEdge.

[12] R. Gurdeep Singh and C. Scholliers, "Warduino: a dynamic webassembly virtual machine for programming microcontrollers," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, 2019, pp. 27–36.

[13] K. Zandberg and E. Baccelli, "Femto-containers: Devops on microcontrollers with lightweight virtualization & isolation for iot software modules," *arXiv preprint arXiv:2106.12553*, 2021.

[14] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An embedded trusted runtime for webassembly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 205–216.

[15] 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.

[16] D. Wang, B. Jiang, and W. Chan, "Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv preprint arXiv:2007.15510*, 2020.

[17] 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*, 2022, pp. 703–715.

[18] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of {WebAssembly} vs. native code," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 107–120.

[19] A. Jangda, B. Powers, A. Guha, and E. Berger, "Mind the gap: Analyzing the performance of webassembly vs. native code," *arXiv preprint arXiv:1901.09056*, 2019.

[20] W. Wang, "Empowering web applications with webassembly: Are we there yet?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 1301–1305.

[21] 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.

[22] J. De Macedo, R. Abreu, R. Pereira, and J. Saraiva, "Webassembly versus javascript: Energy and runtime performance," in *2022 International Conference on ICT for Sustainability (ICT4S)*. IEEE, 2022, pp. 24–34.

[23] ——, "On the runtime and energy performance of webassembly: Is webassembly superior to javascript yet?" in *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. IEEE, 2021, pp. 255–262.

[24] B. Spies and M. Mock, "An evaluation of webassembly in non-web environments," in *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE, 2021, pp. 1–10.

[25] *Awesome WebAssembly Runtimes*, https://github.com/appcypher/awesome-wasm-runtimes.

[26] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[27] J. S. Carlson and K. H. Wiedl, "Toward a differential testing approach: Testing-the-limits employing the raven matrices," *Intelligence*, vol. 3, no. 4, pp. 323–344, 1979.

[28] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007, pp. 549–552.

[29] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 621–631.

[30] *Wasmer*, https://github.com/wasmerio/wasmer.

[31] *Wasmtime*, https://github.com/bytecodealliance/wasmtime.

[32] *Wasm3*, https://github.com/wasm3/wasm3.

[33] *WebAssembly Micro Runtime*, https://github.com/bytecodealliance/wasm-micro-runtime.

[34] *test-suite Guide*, https://llvm.org/docs/TestSuiteGuide.html.

[35] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011, pp. 301–312.

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

[37] L. Clark, "Standardizing wasi: A system interface to run webassembly outside the web," *Mozilla Hacks–the Web developer blog*, 2019.

[38] *Cloud Native Computing Foundation*, https://www.cncf.io/.

[39] *Bytecode Alliance*, https://bytecodealliance.org/.

[40] *Secure & lightweight microservice with a database backend*, https://github.com/second-state/microservice-rust-mysql.

[41] *ab - Apache HTTP server benchmarking tool*, https://httpd.apache.org/docs/2.4/programs/ab.html.

[42] *perf: Linux profiling with performance counters*, https://perf.wiki.kernel.org/index.php/Main_Page.

[43] *PolyBench/C - the Polyhedral Benchmark suite*, https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/.

[44] *Cranelift Code Generator*, https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

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

[46] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski, "Webassembly modules as lightweight containers for liquid iot applications," in *Web Engineering: 21st International Conference, ICWE 2021, Biarritz, France, May 18–21, 2021, Proceedings*. Springer, 2021, pp. 328–336.

[47] V. Kjorveziroski, S. Filiposka, and A. Mishev, "Evaluating webassembly for orchestrated deployment of serverless functions," in *2022 30th Telecommunications Forum (TELFOR)*. IEEE, 2022, pp. 1–4.

[48] W. Huang and M. Paradies, "An evaluation of webassembly and ebpf as offloading mechanisms in the context of computational storage," *arXiv preprint arXiv:2111.01947*, 2021.

[49] M. Nurul-Hoque and K. A. Harras, "Nomad: Cross-platform computational offloading and migration in femtoclouds using webassembly," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 168–178.

[50] 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*. IEEE, 2021, pp. 1–10.

[51] 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)*. IEEE, 2020, pp. 1–4.

[52] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[53] G. Barany, "Finding missed compiler optimizations by differential testing," in *Proceedings of the 27th international conference on compiler construction*, 2018, pp. 82–92.

[54] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[55] C. Kästner, "Differential testing for variational analyses: Experience from developing kconfigreader," *arXiv preprint arXiv:1706.09357*, 2017.

[56] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[57] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[58] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for jvm testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1133–1144.

[59] T. Brennan, S. Saha, and T. Bultan, "Jvm fuzzing for jit-induced side-channel detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1011–1023.

[60] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 114–129.

[61] Y. Chen and Z. Su, "Guided differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 793–804.

[62] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*. IEEE, 2017, pp. 615–632.

[63] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[64] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[65] J. Guo, Y. Zhao, H. Song, and Y. Jiang, "Coverage guided differential adversarial testing of deep learning systems," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 933–942, 2020.