Distinguishability-guided Test Program Generation for WebAssembly Runtime Performance Testing

Shuyao Jiang*, Ruiying Zeng†, 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*
syjiang21@cse.cuhk.edu.hk, ryzeng22@m.fudan.edu.cn, zyf@fudan.edu.cn, lyu@cse.cuhk.edu.hk

***Abstract*—WebAssembly (Wasm) is a binary instruction format designed as a portable compilation target, which has been widely used on both the web and server sides in recent years. As high performance is a critical design goal of Wasm, it is essential to conduct performance testing for Wasm runtimes. However, existing research on Wasm runtime performance testing still suffers from insufficient high-quality test programs. To solve this problem, we propose a novel test program generation approach *WarpGen*. It first extracts code snippets from historical issue-triggering test programs as initial operators, then inserts an operator into a seed program to synthesize a new test program. To verify the quality of generated programs, we propose an indicator called *distinguishability*, which refers to the ability of a test program to distinguish abnormal performance of specific Wasm runtimes. We apply *WarpGen* for performance testing on four Wasm runtimes and verify its effectiveness compared with baseline approaches. In particular, *WarpGen* has identified seven new performance issues in three Wasm runtimes.**

***Index Terms*—WebAssembly, performance testing, test program generation**

I. Introduction

WebAssembly (Wasm) [1] is a binary instruction format designed as a portable compilation target for programming languages (*e.g.*, C, C++ and Rust). Wasm is designed to execute at native speed, provide type and memory safety, and be portable across different languages and platforms. These advantages of Wasm make it popular in a wide range of fields, including web and non-web environments [2]–[6]. Four major browsers (*i.e.*, Chrome, Firefox, Safari, and Edge) have supported Wasm to enable higher-performance web applications [7]. Wasm has also been increasingly used in many server-side applications (*e.g.*, cloud service [8]–[10], microcontrollers [11], [12], and smart contracts [13]–[15]).

High performance is a critical design goal of Wasm, so it is highly essential to conduct Wasm runtime performance testing, especially for server-side Wasm applications. Unlike web applications running on well-developed browser engines, server-side Wasm applications need to run in standalone Wasm runtimes (*e.g.*, WasmEdge [16]), which are still immature and more likely to cause performance issues (*i.e.*, abnormal latency). Jiang *et al.* [17] recently pointed out the severe impact of performance issues on server-side Wasm applications. They found that a 30ms-latency will result in up to 50% drop of service throughput in a real-world Wasm microservice [18]. They further proposed a differential testing approach *WarpDiff* to identify some performance issues in existing standalone Wasm runtimes. The results indicate that performance issues are common in many Wasm runtimes, which can significantly threaten the reliability of Wasm applications.

However, state-of-the-art research on Wasm runtime performance testing still suffers from insufficient high-quality test programs. *WarpDiff* only used a small benchmark (*i.e.*, 123 programs from the LLVM Test Suite [19]) for testing, so the identified issues are very limited. To further uncover more performance issues, it is necessary to collect or generate more test programs. Unfortunately, existing test suites and test program generation approaches are mainly targeted at finding functional bugs (*i.e.*, software errors that cause wrong execution results) in specific software (*e.g.*, Csmith [20], a popular program generator for C compiler testing). They are unsuitable for applying directly to Wasm runtime performance testing since the test programs are not tailored for triggering Wasm runtime performance issues. Without high-quality (*i.e.*, issue-triggering) test programs, much time and manual effort will be spent on case filtering and issue verification, dramatically hurting testing efficiency.

To solve this problem, it is critical to design an *efficient* test program generation approach for Wasm runtime performance testing. However, it is extremely challenging to achieve this goal. Specifically, we need to address two main challenges. The first challenge is the lack of prior knowledge about what programs tend to trigger performance issues in Wasm runtimes. Unlike other testing tasks (*e.g.*, compiler testing and JVM testing) that have been widely studied for decades, Wasm runtime performance testing is still in the early stages of practice. To the best of our knowledge, the only current experience with Wasm runtime performance is the few cases reported by *WarpDiff*, which is too limited for test program generation. The second challenge is how to verify the quality of newly generated test programs. We treat test programs that trigger performance issues in Wasm runtimes as high-quality test programs. However, it is infeasible to verify whether each generated program triggers performance issues manually. We need to design an indicator to automatically verify the quality of test programs and keep high-quality ones.

In this paper, we propose ***WarpGen***, a novel test program generation approach for Wasm runtime performance testing. To address the first challenge, *WarpGen* adopts the idea of

✉ Yangfan Zhou is the corresponding author.

history-driven test program generation. The insight is that historical issue-triggering test programs contain code snippets that help detect new issues. This idea has been verified in many testing tasks, including C compiler testing [21], [22] and JVM testing [23]. Based on this idea, our limited prior knowledge about Wasm runtime performance can still provide insights for finding new issues. Specifically, *WarpGen* first extracts code snippets from abnormal cases (*i.e.*, the test programs that have revealed Wasm runtime performance issues) reported by *WarpDiff*. Then, *WarpGen* integrates these code snippets with different contexts to generate new test programs for triggering new performance issues. However, not all generated programs can achieve this goal. Therefore, to address the second challenge, we propose an indicator, namely ***distinguishability***, to measure the quality of the generated test programs. Conceptually, the *distinguishability* refers to the ability of a test program to distinguish the abnormal performance of some Wasm runtimes. To formalize it, we draw inspiration from the idea of *WarpDiff*, that is, the execution time of the same test case on different Wasm runtimes should follow a stable ratio (*i.e.*, *oracle ratio*). So, we can identify an abnormal case whose execution time ratio significantly deviates from the *oracle ratio*. Hence, we formalize the *distinguishability* of a test program as the distance between the vector of its execution time ratio on several Wasm runtimes and the vector of the *oracle ratio*, called *dist score*. *WarpGen* uses the *dist score* to measure the quality of a test program and guide the whole process of program generation.

Based on the above insights, the workflow of *WarpGen* is designed as follows: For preparation, *WarpGen* extracts a series of code snippets (called *operators*) from the abnormal cases reported by *WarpDiff* to construct the operator pool, and collects some seed programs to construct the seed pool. In the initial iteration, for each operator, *WarpGen* inserts it into a seed program to generate a new test program, then executes the Wasm code on several Wasm runtimes and calculates its *dist score*. After the initial iteration, *WarpGen* treats the test programs with top N *dist score* as *distinguishable programs*, then extracts operators from these programs and adds them to the operator pool. To improve the efficiency of further iterations, *WarpGen* assigns a *penalty* with an initial zero value for each operator. In the follow-up iterations, each time, *WarpGen* randomly selects an operator and a seed program to generate a new program, then calculates its *dist score* after execution. If the *dist score* exceeds the previous top N, *WarpGen* will mark it as a new *distinguishable program*. Also, *WarpGen* will update the top N scores and reset the *penalty* of the selected operator. Otherwise, the *penalty* of the selected operator will increase by one. When the *penalty* of an operator accumulates to M, it will be removed from the operator pool. Thus, *WarpGen* can generate programs with an increasing trend of *dist score*.

For evaluation, we conducted a series of experiments. We used *WarpGen* to generate C programs for performance testing on four popular Wasm runtimes: Wasmer [24], Wasmtime [25], WasmEdge [16], and WAMR [26]. To evaluate the efficiency of *WarpGen* to generate high-quality test programs, we collected the statistics of *dist score* during the program generation. We found that the *dist score* of *distinguishable programs* achieved the optimal result quickly, which means that *WarpGen* can generate high-quality test programs efficiently. To further verify the effectiveness of the *distinguishability*-guided design, we compared *WarpGen* with a random approach Csmith and a non-guided approach *WarpGen-base*. We compared the *dist score* of the generated test programs from the three approaches, and we found that *WarpGen* can achieve the best overall performance. Finally, *WarpGen* identified seven previously unknown performance issues in three Wasm runtimes. The results verify the effectiveness of *WarpGen* for Wasm runtime performance testing.

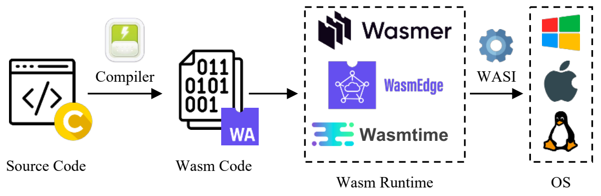


Fig. 1. The workflow of Wasm compilation and execution.

In summary, this work makes the following contributions:

- **Direction.** We conduct the first study on test program generation for Wasm runtime performance testing, and we propose a novel indicator (*i.e.*, *distinguishability*) to guide test program generation.
- **Approach.** We design and implement a *distinguishability*-guided test program generation approach *WarpGen*, which aims to efficiently generate test programs that trigger performance issues in Wasm runtimes.
- **Empirical study.** We apply *WarpGen* for performance testing on four real-world Wasm runtimes and verify its effectiveness compared with other approaches. *WarpGen* has identified seven previously unknown performance issues in three Wasm runtimes.

We make our source code and experiment data available at https://figshare.com/s/9d1b5e43b80029d14c42.

II. Background

A. Wasm Runtime

Wasm is designed as a compilation target for high-level programming languages (*e.g.*, C/C++). The typical workflow of deploying a Wasm application is first to compile the source program into Wasm bytecodes, then execute the program on a Wasm runtime, as shown in Figure 1. The Wasm runtime plays a vital role in this workflow. It is a virtual machine that translates Wasm binary instructions to native machine code for execution. It provides a memory-safe, sandboxed execution environment for Wasm, working as an intermediary between the Wasm application and the operating system (OS). The mechanism of Wasm runtime enables Wasm to be portable across different platforms.

For Wasm applications in different environments, the implementation of Wasm runtimes is different. In the web environment, the Wasm bytecodes run in the browser with the help of JavaScript glue code. The glue code would call into the browser engine (*e.g.*, V8 [27], SpiderMonkey [28]), which would then talk to the operating system. However, in the non-web (server-side) environment, Wasm needs a particular interface, the WebAssembly system interface (WASI) [29], to access the resources in the operating system. So, server-side Wasm applications need to run in a standalone Wasm runtime with WASI support. With the increasing applications of Wasm in the non-web environment, many standalone Wasm runtimes are emerging. There are currently more than 30 standalone Wasm runtime implementations on GitHub [30]. Some representative standalone Wasm runtimes include Wasmer [24], Wasmtime [25], WasmEdge [16], WAMR [26], etc.

Unlike the major browser engines that are well-developed, existing standalone Wasm runtimes are still in the early stages of development. Most standalone Wasm runtimes do not have mature optimization mechanisms, which may cause performance issues (*i.e.*, abnormal latency) during execution. Compared to functional issues, performance issues in Wasm runtimes are more likely to be overlooked but can lead to severe consequences. Jiang *et al.* [17] have found that a short latency of 30ms will result in up to 50% drop of service throughput in a real-world Wasm microservice [18]. Thus, performance testing is a critical task for Wasm runtimes.

B. Performance Testing for Wasm Runtimes

Although performance testing is critical for Wasm runtimes, research in this area is still limited. Existing works about Wasm performance mainly study the systematic performance gaps between Wasm and native code or JavaScript [31]–[34], but little focus on *performance issues* in Wasm runtimes. One state-of-the-art research towards Wasm runtime performance issues is *WarpDiff* [17], a differential testing approach for identifying performance issues in Wasm runtimes.

WarpDiff aims to solve the problem of lacking test oracle in performance testing on Wasm runtimes, *i.e.*, how to identify the occurrence of performance issues. Unlike functional issues with a clear test oracle (*e.g.*, wrong execution results), performance issues are hard to detect since there is no ground truth of the performance indicator (*i.e.*, the execution time of a test case). To solve this problem, *WarpDiff* proposed a new idea for performance testing: *The execution time of the same test case on different Wasm runtimes should follow a stable ratio (i.e., **oracle ratio**) under normal circumstances.* For each test case, *WarpDiff* collected its execution time on several Wasm runtimes to construct a vector representing its execution time ratio on these runtimes. Then, it got the vector of the *oracle ratio* based on the average execution time ratio of all the test cases. *WarpDiff* compared the distance between the two normalized vectors for each test case, then identified an abnormal case in which the execution time ratio significantly deviates from the *oracle ratio*. Based on this idea, *WarpDiff* conducted performance testing on several Wasm runtimes using 123 test programs from the LLVM Test Suite [19] and identified seven performance issues in four runtimes.

Although *WarpDiff* has addressed the lack of test oracle for Wasm runtime performance testing, it still suffers from another challenge: insufficient high-quality test programs. *WarpDiff* only used a small benchmark for testing. Thus, the number of identified performance issues is also limited. To discover more performance issues in Wasm runtimes, it is necessary to conduct large-scale performance testing using more test programs. However, using randomly selected programs for performance testing is inefficient, since most test cases cannot trigger performance issues in Wasm runtimes. Without high-quality test programs, we must spend a lot of time and manual effort on issue verification. Therefore, to improve the testing efficiency, we aim to design a test program generation approach that produces high-quality (*i.e.*, issue-triggering) test programs for Wasm runtime performance testing.

III. Approach

Although test program generation has been widely studied, there is still a lack of test program generation approach targeting at Wasm runtime performance testing. There are two unique challenges in this task: 1) *We lack sufficient prior knowledge about what kinds of programs can trigger performance issues.* Compared with other testing tasks (*e.g.*, compiler testing) that have been studied for many years, our experience on Wasm runtime performance is too limited for test program generation. 2) *It is difficult to verify the quality of generated test programs.* The key criterion for high-quality test programs is that they can trigger performance issues in Wasm runtimes, *i.e.*, they can distinguish the abnormal performance of some Wasm runtimes. However, it is infeasible to check each test program manually. We need an indicator for automatic quality verification of test programs.

To address the above challenges, we propose ***WarpGen***, a novel test program generation approach for Wasm runtime performance testing. The design of *WarpGen* contains two key insights: 1) *Historical issue-triggering test programs contain information that helps detect new issues.* This idea has been verified in many previous studies, including compiler testing [21], [22] and JVM testing [23]. Although our historical experience towards Wasm runtime performance issues is limited (only a few known issues reported by *WarpDiff*), we can still extract useful information from those previous issue-triggering test programs to generate new test programs. 2) *The test oracle proposed by WarpDiff can inspire test program quality verification. WarpDiff* has verified that Wasm runtime performance issues can be revealed from those test programs in which the execution time ratio significantly deviates from the *oracle ratio*. In other words, such test programs can distinguish the abnormal performance of certain Wasm runtimes and should be considered high-quality test programs. Therefore, we propose an indicator called ***distinguishability*** to measure the quality of the generated programs. We formalize the *distinguishability* of a test program as the distance between the two vectors of its execution time ratio and the *oracle ratio*,

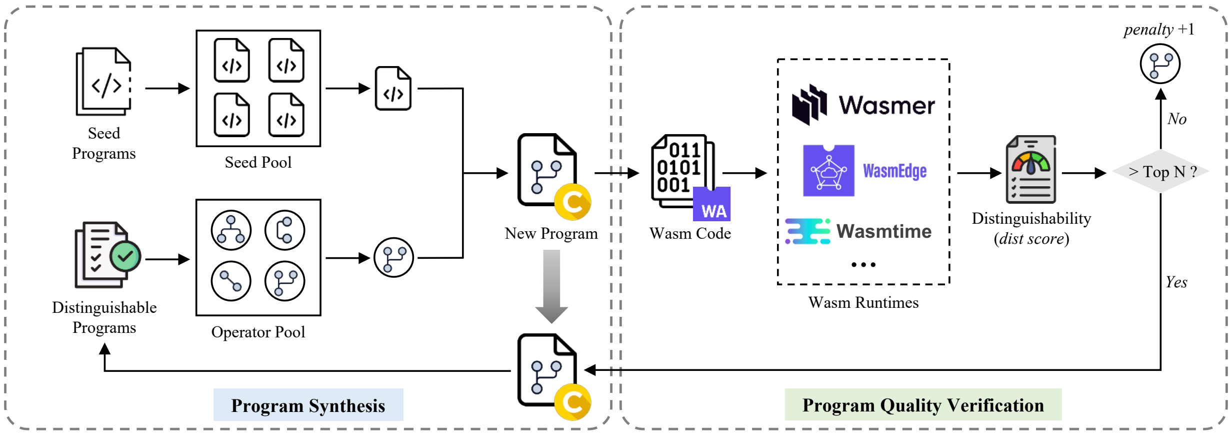


Fig. 2. The framework of *WarpGen*.

called *dist score*. *WarpGen* uses the *dist score* to guide the test program generation process.

Figure 2 shows the framework of *WarpGen*. Overall, *WarpGen* contains two key modules: program synthesis and program quality verification. As a preparation, *WarpGen* first extracts a series of code snippets (called *operators*) from the historical issue-triggering test programs reported by *WarpDiff* to initialize the operator pool; then collects some seed programs to initialize the seed pool (**Section III-A**). To generate a new test program, *WarpGen* selects an operator to insert into a seed program. During the insertion, we need to solve two critical problems: *insertion point selection* and *variable dependency handling* (**Section III-B**). The iteration process is designed as follows: In the initial iteration, *WarpGen* first generates a new test program for each initial operator, then collects the test programs with top N *dist score* as *distinguishable programs* for updating the operator pool. In the follow-up iterations, for a newly generated test program, *WarpGen* checks whether its *dist score* exceeds the previous top N for further updates (**Section III-C**). Next, we will elaborate on the design and implementation details of *WarpGen*.

A. Data Preprocessing

In this subsection, we first introduce how *WarpGen* preprocesses data for test program generation, including *operator extraction* and *seed profiling*. Since the operators and seeds are from different programs, a big challenge of synthesizing a new program is to ensure the validity of the synthesized program. The program validity includes two aspects:

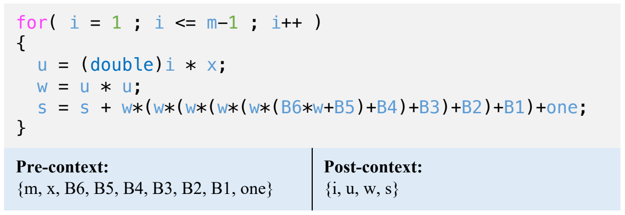
- **Syntax Validity:** The synthesized program should conform to the syntax rules and be able to pass compilation.
- **Insertion Validity:** The inserted operator should affect the behavior of the seed program.

To address this challenge, the key operation in data preprocessing is to record the related contexts for program synthesis, which will be explained later.

Operator Extraction. To extract operators (code snippets) from a source program, we first need to determine the operator granularity, *e.g.*, line, block, or function. Fine-grained code snippets (*e.g.*, code line) have simple context but usually cannot express complete functionality. On the contrary, coarse-grained code snippets (*e.g.*, function) may have a complex context that is difficult to process during program synthesis. Therefore, *WarpGen* adopts block granularity for extraction since it can achieve a trade-off between functional completeness and processing difficulty. Specifically, *WarpGen* extracts the following four types of operators:

- **Sequential Operator:** A code block containing sequential statements without branches and loops.
- **Branching Operator:** A code block containing conditional branching statements (*i.e.*, `if`, `else`, and `else if`), including conditions and corresponding bodies.
- **Looping Operator:** A code block containing looping statements (*i.e.*, `for`, `while`, and `do-while`), including loop conditions and loop bodies.
- **Mixed Operator:** A code block containing a combination of the above three operators.

For the implementation of operator extraction, *WarpGen* implements a customized Clang tool named `extract-blocks` based on the LLVM Project [35], a modular toolchain for C-like languages. For each input source program, *WarpGen* first transfers the program to an Abstract Syntax Tree (AST). The AST describes the syntactical structure of the source program. Each node in the AST refers to a statement (`Stmt`) or a declaration (`Decl`) in the program, and each edge refers to the logic between the two nodes. To extract operators from the source program, *WarpGen* traverses its AST to collect `Stmt` with specific classes. Specifically, to extract branching operators, *WarpGen* collects statements with the class of `IfStmt`. To extract looping operators, *WarpGen* collects statements with the classes of `ForStmt`, `WhileStmt`, and `DoStmt`. For sequential operators, we cannot collect them directly since there is no class representing sequential statements in AST. To solve this problem, *WarpGen* first collects statements with the class of `CompoundStmt`, which refers to a block surrounded by '{}'. Among these blocks, those without branching statements



```
for( i = 1 ; i <= m-1 ; i++ )
{
  u = (double)i * x;
  w = u * u;
  s = s + w*(w*(w*(w*(w*(B6*w+B5)+B4)+B3)+B2)+B1)+one;
}
```

Pre-context:	**Post-context:**
{m, x, B6, B5, B4, B3, B2, B1, one}	{i, u, w, s}

Fig. 3. A *looping operator* and its pre/post-contexts. The type and positions of the variables are omitted.

and looping statements are treated as sequential operators, and the rest are treated as mixed operators.

As mentioned above, a critical challenge in program synthesis is to ensure the *syntax validity* and *insertion validity* of the new program. To achieve this goal, during operator extraction, *WarpGen* records the following two contexts for each operator:

- **Pre-context:** All used variables (excluding variables declared and assigned in this block) and called functions (excluding standard library functions) in this block.
- **Post-context:** All variables assigned in this block (excluding variables declared in this block).

On the one hand, the variables and functions in the pre-context would not be found in the seed program and should be replaced accordingly to ensure the *syntax validity*. We exclude the declared variables and the standard library functions because they would not cause syntax errors in the synthesized program. On the other hand, the variables in the post-context should be replaced by the variables that would be used after the insertion point of the operator. Only in this way can the inserted operator affect the behavior of the seed program to ensure the *insertion validity*. To obtain the two contexts of an operator, *WarpGen* first traverses all its used variables and functions in the AST (nodes of type `DeclRefExpr`). For each variable, *WarpGen* checks whether it is on the left side of an `AssignmentOperator` (*e.g.*, =, +=, and -=). If so, the variable will be added to the post-context. Otherwise, it will be added to the pre-context. The functions (excluding standard library functions, *e.g.*, `printf`) will be directly added to the pre-context. For each variable and function, *WarpGen* records its name, type, and all positions.

To better explain how *WarpGen* obtains the two contexts, we provide an example, as shown in Figure 3. This example shows a looping operator extracted by *WarpGen*. In the source code of this operator, four variables (*i.e.*, `i`, `u`, `w`, and `s`) are assigned new values. To ensure the *insertion validity*, the values of these four variables should be used after the insertion point of this operator. Therefore, *WarpGen* adds them to the post-context of this operator. Other variables used but not declared in this code (*i.e.*, `m`, `x`, `B6`, `B5`, `B4`, `B3`, `B2`, `B1`, and `one`) are added into the pre-context. They should be replaced or redefined to ensure the *syntax validity*.

After initial operator extraction, *WarpGen* will obtain some initial operators from the historical issue-triggering test programs. Each operator is saved as a JSON file, recording its source code and the pre/post-contexts. During the iteration process of *WarpGen*, it will further extract new operators from new *distinguishable programs* to update the operator pool.

Seed Profiling. To provide diverse contexts for synthesized programs, *WarpGen* picks some C programs different from the source programs of the operators to construct the seed pool. The selection of seed programs can be customized. To ensure the *syntax validity* and *insertion validity* of the synthesized programs, *WarpGen* also needs to profile the seed programs.

Specifically, *WarpGen* profiles the *variable usage* and *code coverage* for each seed program. The *variable usage* is used to ensure both *syntax validity* and *insertion validity*. Given a seed program, *WarpGen* first traverses its AST to collect all the declared variables and functions (nodes of type `VarDecl`). For each variable (including function), *WarpGen* records the positions (code line number) where the variable was first defined and last used. The purpose of recording the two positions is: During program synthesis, the variables for replacing the pre-context of the operator should be defined before the insertion point, and the variables for replacing the post-context of the operator should be used after the insertion point. *WarpGen* implements a Clang tool named `ues-define-tag` to obtain the *variable usage*. On the other hand, the *code coverage* is used to ensure the *insertion validity*. To make a valid insertion, the insertion point of the operator should always be covered during program execution, *i.e.*, the inserted operator can always be executed. Therefore, given a seed program, *WarpGen* also executes it and profiles its *code coverage* using `llvm-profdata` and `llvm-cov` [35], then records all the covered code lines during execution.

B. Program Synthesis

In this subsection, we describe how *WarpGen* synthesizes a new test program given an operator and a seed program. During program synthesis, *WarpGen* needs to resolve two critical problems: *insertion point selection* and *variable dependency*. *WarpGen* resolves these two problems based on the preprocessing information described in Section III-A. The specific process is as follows:

Insertion Point Selection. For the same seed program, the valid insertion points. (*i.e.*, the synthesized program can satisfy the *syntax validity* and *insertion validity*) of different operators are different, since each operator has unique pre/post-contexts. Therefore, before inserting an operator, *WarpGen* first needs to search valid insertion points for this operator. To this end, *WarpGen* traverses the code lines recorded in the *code coverage* information of the seed program (described in Section III-A) and checks which lines can be valid insertion points. Specifically, when *WarpGen* visits a specific code line, it collects the current context of the seed program, including all the local variables that have been defined at this point and all the global variables. This current context is obtained based on the *variable usage* of the seed program. Then, *WarpGen* reads the pre/post-contexts of the operator to check whether the current context of the seed program can meet the requirements for replacing the pre/post-contexts of the operator. For each variable (including function) to be replaced, *WarpGen* first checks if there is a reusable local/global variable in the seed program. If not, *WarpGen* will check whether it is feasible to define a new variable here. In this way, *WarpGen* finds all valid insertion points for the operator and randomly selects a point for insertion.

Variable Dependency Handling. After selecting an insertion point, *WarpGen* then handles the variable dependency of the inserted operator. As mentioned above, for a variable (including function) x to be replaced, *WarpGen* first tries to find a reusable variable y in the seed program. The reusable variable y should satisfy the following conditions: First, The type of y must be the same as x. Meanwhile, for a variable in the post-context of the operator, the last used position of y should be after the current insertion point. If such a variable y is found, *WarpGen* will replace x at all positions in the operator with y. If multiple variables satisfy the condition, *WarpGen* will randomly select one for replacement.

However, if no reusable variables for x are found in the seed program, *WarpGen* will define a new variable to ensure the validity of the synthesized program. Specifically, *WarpGen* first declares a new variable n of the same type as x, then assigns a value for n. The assigned value is randomly selected from a predefined variable pool. *WarpGen* has implemented the function of defining new variables for all basic data types, array and pointer types in C. After defining the variable n, *WarpGen* will replace x at all positions in the operator with n, then insert the variable definition statements and the modified operator into the seed program to synthesize the new program. *WarpGen* implements a Clang tool named `insert` to achieve the above program synthesis process.

C. Iteration Process

In this subsection, we elaborate on the whole iteration process of *WarpGen*. In particular, we will explain how *WarpGen* adopts *distinguishability* (*dist score*) as the program quality indicator to guide the test program generation process.

Based on our design insights, those test programs that can distinguish the abnormal performance of certain Wasm runtimes are treated as high-quality test programs. Such test programs are manifested as that, their execution time ratio on several Wasm runtimes significantly deviates from the *oracle ratio*. Therefore, we propose an indicator named *distinguishability* to measure the quality of test programs. We formalize the *distinguishability* of a test program as *dist score*, which is defined as follows:

The ***distinguishability (dist score)*** *of a test program is the Euclidean distance between the normalized vector of its execution time ratio (on Wasm runtimes to be tested) and the normalized vector of the oracle ratio.*

For example, if the execution time of a test program x on three Wasm runtimes are 1s, 2s, and 3s, then its original execution time vector can be represented as $[1, 2, 3]$. However, the original vectors of different test programs are

Algorithm 1: Iteration Process of *WarpGen*

```
Input  : seedPool, opPool, N, M, k
Output : top N distinguishable programs
   // Initial Iteration
1  foreach operator op in opPool do
2      seed ← a random seed in seedPool;
3      synProgram ← synthesizeProgram(op, seed);
4      distScore ← getDistScore(synProgram);
5  topScoreSet ← top N distScore values;
6  topProgramSet ← programs with top N distScore values;
7  foreach program p in topProgramSet do
8      opPool ← opPool ∪ extractOps(p);
9  foreach operator op in opPool do  penalty_op ← 0 ;
   // Follow-up Iterations
10 while program numbers < k AND opPool ≠ ∅ do
11     op, seed ← a random value in opPool, seedPool;
12     synProgram ← synthesizeProgram(op, seed);
13     distScore ← getDistScore(synProgram);
14     if distScore > min(topScoreSet) then
15         opPool ← opPool ∪ extractOps(synProgram);
16         update(topScoreSet, topProgramSet);
17         penalty_op ← 0;
18     else penalty_op ← penalty_op + 1;
19     if penalty_op == M then opPool ← opPool \ {op};
```

not comparable since the execution time of each program is affected by its features. Therefore, we need to normalize the original execution time vector for each program. In this way, the test programs with the same execution time ratio will have the same normalized vectors. For example, the original vector $[10, 20, 30]$ of test program y will be normalized to $[0.17, 0.33, 0.5]$ (*L1-normalization*), the same as that of x. If the normalized vector of the *oracle ratio* is $[0.2, 0.4, 0.4]$, then the *dist score* of x (and also y) is 0.12, the Euclidean distance between the two normalized vectors. The higher *dist score* means the test program is more capable of distinguishing abnormal performance in specific Wasm runtimes.

Next, we will describe the whole iteration process of *WarpGen*, as shown in Algorithm 1. The iteration process aims to identify the top N *distinguishable programs*, *i.e.*, test programs with top N *dist score* (the value of N can be customized). The intention is that the performance of the tested Wasm runtimes deviates more from the expected performance (*i.e.*, *oracle ratio*) when executing these programs, *i.e.*, they can reveal performance issues with high severity.

Initial Iteration. In the initial iteration, *WarpGen* aims to set the initial top N *distinguishable programs*. To this end, *WarpGen* first traverses the initial operator pool. For each operator, *WarpGen* tries to insert it into a randomly selected seed program to synthesize a new test program (based on the strategy described in Section III-B). If an operator cannot be inserted into any seed program successfully, *WarpGen* will remove it from the operator pool. Then, *WarpGen* compiles the synthesized program to Wasm code using Emscripten [36], a typical compiler for compiling C/C++ programs to Wasm. After compilation, *WarpGen* executes the Wasm code on several Wasm runtimes (test objects) and calculates the *dist score* of this test program. Thus, after the initial iteration, *WarpGen* collects the test programs with top N *dist score* as the initial *distinguishable programs*. However, we cannot identify which part of these test programs caused their high *dist score* without manual analysis. A reasonable solution is to extract operators from these test programs for further iterations. Therefore, at this step, *WarpGen* extracts operators from the initial *distinguishable programs* and adds them to the operator pool.

Follow-up Iterations. In follow-up iterations, the goal of *WarpGen* is to continuously find new test programs with higher *dist score* than the previous top N programs. According to our design, the operators in the operator pool will gradually increase during the iteration process, but not every operator is effective for generating high-quality test programs. Random selection of operators is inefficient for generating high-quality test programs. Therefore, to improve the efficiency of test program generation, we design a *penalty mechanism* for operator selection. Before iterations, *WarpGen* assigns a *penalty* with an initial zero value for each operator (including those obtained in the initial iteration) in the operator pool. During the follow-up iterations, each time, *WarpGen* randomly selects an operator and a seed to generate a new test program, then calculates its *dist score* after execution. If the *dist score* exceeds the previous top N scores, *WarpGen* will mark it as a new *distinguishable program*, then extract its operators and add them to the operator pool. Also, *WarpGen* will update the top N scores and reset the *penalty* of the selected operator to zero. Otherwise, the *penalty* of the selected operator will be increased by one. When the *penalty* of an operator is accumulated to M, it will be removed from the operator pool. The value of M can also be customized. After multiple iterations, the operators remaining in the operator pool will be more likely to contribute to test programs with high *dist score*. Finally, the iteration process will stop when the total number of generated test programs reaches the customized value k, or when the operator pool becomes empty.

IV. Evaluation

In this section, we address the following research questions:

- **RQ1:** How efficient is *WarpGen* to generate high-quality test programs?
- **RQ2:** How effective is the *distinguishability*-guided design in *WarpGen*?
- **RQ3:** Can *WarpGen* detect new performance issues in existing Wasm runtimes?

A. Experiment Settings

Test objects. In our experiments, we selected four standalone Wasm runtimes, *i.e.*, Wasmer [24], Wasmtime [25], WasmEdge [16], and WAMR [26], as test objects. Table I shows their information. The considerations for selecting these runtimes include two aspects. On the one hand, they are all representative Wasm runtimes with high *popularity* (*i.e.*, with the top number of GitHub stars) and *activity* (*i.e.*, the latest commit is within one month). On the other hand, *WarpDiff* has reported that all these four Wasm runtimes contain performance issues, which need our further attention. We selected the latest version of each runtime for testing and tested all the runtimes in AOT (Ahead-of-Time) mode.

TABLE I
Wasm runtimes as test objects.

Runtime	Language	#Stars	#Commits	Version
Wasmer	Rust	16.7k	16.3k	4.2.3
Wasmtime	Rust	13.5k	12.5k	cli 15.0.0
WasmEdge	C/C++	7.2k	2.9k	0.13.5
WAMR	C/C++	4.2k	1.5k	1.2.3

Initial Operators. Based on the first insight of *WarpGen*, We collected historical issue-triggering test programs to initialize the operator pool. Specifically, we collected the top 20 abnormal test programs reported by *WarpDiff* for operator extraction, since they are currently the only source related to Wasm runtime performance issues. These programs are from four Benchmarks (*i.e.*, BenchmarkGame, Misc, Shootout, and Polybench) in the LLVM Test Suite [19] and written in C. Finally, *WarpGen* obtained 271 initial operators from these programs. We show the details of these operators in our supplementary data due to space limitations.

Seed Programs. Since the initial operators are code snippets from C programs, the seeds should also be C programs. In our experiments, we selected 100 C programs randomly generated by Csmith [20] as seeds. Csmith is one of the most popular random program generators that can generate C programs free of undefined behaviors, which is suitable for providing seed programs in our task. It is worth noting that *WarpGen* accepts any C program as a seed. So, there are also many other ways of setting seed programs, and we just chose one appropriate method for experiments.

Parameters. There are some customized parameters in *WarpGen* design. First, *WarpGen* needs to maintain the top N values of *dist score* and the corresponding test programs during iteration. We set the value of N to 20 in experiments, because the performance deviation shown by the cases after the top 20 is less significant according to the data reported by *WarpDiff*. Second, we set the *penalty* threshold M for operator removal to 5, which can achieve a trade-off between iteration efficiency and test program quality based on our observation during the experiments. Then, we set the total number k of generated test programs to 1,000 in our experiments. In addition, we calculated the *oracle ratio* based on the average execution time ratio of the seed programs, which represents the normal performance of the tested Wasm runtimes.

Compared Approaches. To answer RQ2, we compared *WarpGen* with two baseline approaches. The first one is random programs (different from the seeds) generated by Csmith. The second one is a simplified version of *WarpGen* (named *WarpGen-base*), where the *distinguishability*-guided design is removed. *WarpGen-base* generates a test program just by randomly selecting an operator and a seed for program synthesis. It will not update the operator pool and the top program set. We evaluated the quality of the test programs under these three approaches to verify the effectiveness of the

TABLE II
STATISTICS OF TOP 20 *dist score* WHEN *WarpGen* GENERATED DIFFERENT NUMBERS OF TEST PROGRAMS.

#Programs	20	50	80	110	140	170	200	230	260	290	320	350	380	410	436
Minimal	0.023	0.149	0.208	0.259	0.454	0.454	0.454	0.454	0.454	0.454	0.454	0.458	0.459	0.459	0.459
Average	0.145	0.226	0.266	0.392	0.462	0.462	0.462	0.462	0.462	0.462	0.462	0.479	0.481	0.481	0.481

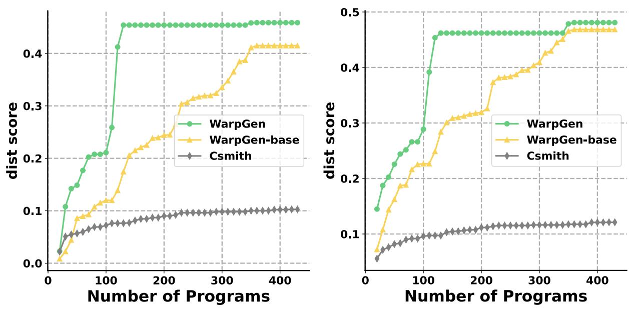
distinguishability-guided design in *WarpGen*.

Implementation and Environment. We implemented *WarpGen* as a Java project, which integrates all of our self-implemented Clang tools (*i.e.*, `extract-blocks`, `use-define-tag`, and `insert`, written in C++) and other necessary tools for testing (*i.e.*, Wasm compiler and runtimes). The parameters and the data paths can be easily modified in the configuration file. We also implemented a complete error handling and checkpoint mechanism for *WarpGen*. The project is implemented by over 5k lines of code, including 2k lines of Java and 3k lines of C++. All our experiments are conducted on a server with a 4-core Intel(R) Xeon(R) E5-2686 v4 CPU @ 2.30GHz and 16GB RAM. The operating system is 64-bit Ubuntu 20.04.6 LTS with the Linux kernel version 5.15.0.

B. RQ1: Efficiency of WarpGen

To evaluate the efficiency of *WarpGen* to generate high-quality test programs, we collected the statistics of *dist score* for each generated test program during the iteration process of *WarpGen*. For analysis, we counted the minimal of the top 20 *dist score* (*i.e.*, the 20th *dist score*) and the average of the top 20 *dist score* with the number of generated test programs increased. We reported these data because the goal of *WarpGen* is to find *distinguishable programs* with top N *dist score*. Since we set the value of N to 20 in our experiments, we focused on the top 20 *dist score*, which indicates the quality of the target test programs of *WarpGen*. We reported the minimal of the top 20 *dist score* since it is the threshold for updating *distinguishable programs*. The change in this value reflects the growth trend of the test program quality. We also reported the average of the top 20 *dist score*, which reflects the average quality of the *distinguishable programs*.

The results are shown in Table II. We report the results for every 30 test programs generated by *WarpGen*. We finally got 436 generated programs in our experiments, since the operator pool became empty before the number of generated test programs reached the set value (early stop). It is worth mentioning that the early stop does not mean the effectiveness of *WarpGen* is not good. On the contrary, it means that *WarpGen* can achieve the optimal result quickly without generating a large amount of test programs. We can observe from Table II that, in the early stage of test program generation (*i.e.*, when the number of generated programs is less than 150), the minimal and the average of top 20 *dist score* grew quickly. When *WarpGen* generated 140 test programs, the minimal of the top 20 *dist score* had reached 0.454, approaching the final optimal result of 0.459. The results indicate that *WarpGen* can generate high-quality test programs for Wasm runtime performance testing with high efficiency.



(a) Minimal of top 20 (b) Average of top 20

Fig. 4. Comparison of top 20 *dist score* from different approaches.

C. RQ2: Effectiveness of Guidance

To further study the effectiveness of *WarpGen*, especially the *distinguishability*-guided design, we compared *WarpGen* with two baseline approaches, Csmith [20] and *WarpGen-base*. Csmith is a representative random C program generator. We treated Csimth as a baseline to evaluate the effectiveness of *WarpGen* relative to random approaches. *WarpGen-base* is a simplified version of *WarpGen* without guidance in the program generation process. We implemented *WarpGen-base* to evaluate the effectiveness of the *distinguishability*-guided design in *WarpGen*. We applied these two baseline approaches to generate the same number of test programs as *WarpGen*, and compared the change in the top 20 *dist score* from the three approaches. Similar to RQ1, we reported the minimal and the average of the top 20 *dist score*. The comparison results are shown in Figure 4, where the data was sampled for every ten programs generated.

From the comparison results, we can find that *WarpGen* achieved the best performance among the three approaches. On the one hand, the *growth rate* of *WarpGen*'s curve is significantly greater than that of the two baseline approaches in the early stage of program generation. *WarpGen* achieved the near-optimal result quickly (when generating 100+ programs), while *WarpGen-base* did not achieve the optimal result until generating 350+ programs. For Csmith, the top 20 *dist score* changed little during the whole program generation process. This result indicates that *WarpGen* can generate high-quality test programs *more quickly* than the baseline approaches. On the other hand, the *final results* of *WarpGen* are also the best among the three approaches, for both the minimal and the average of the top 20 *dist score*. In particular, the final results

TABLE III
PERFORMANCE ISSUES IDENTIFIED BY *WarpGen*.

ID	Runtime	Scenario	Status
#7731	Wasmtime	Floating-point (FP) arithmetic	Fixed
#7732	Wasmtime	Access of pointers to constant	Confirmed
#7733	Wasmtime	Increment operation in nested loops	Confirmed
#4378	Wasmer	Operations on FP arrays	Fixed
#4379	Wasmer	Call of standard output functions	Fixed
#4380	Wasmer	Access of variable addresses	Fixed
#2938	WAMR	FP arithmetic	Confirmed

of *WarpGen* are four times greater than that of Csmith. This result indicates that *WarpGen* can also generate *higher-quality* test programs than the baseline approaches.

In summary, *WarpGen* can generate test programs of *higher quality* and *more quickly* than the baseline approaches. The comparison between *WarpGen* and Csmith shows the effectiveness of *WarpGen* relative to random approaches. The comparison between *WarpGen* and *WarpGen-base* verifies that the *distinguishability*-guided design is effective for *WarpGen* to generate high-quality test programs with high efficiency.

D. RQ3: New Performance Issues

To verify the ability of *WarpGen* to identify new performance issues in Wasm runtimes, we analyzed the top 20 *distinguishable programs* generated by *WarpGen*. We first need to locate the issue-related runtime for each test program, *i.e.*, determine in which Wasm runtime the abnormal latency occurred. To this end, we adopted the performance issue location method in *WarpDiff*. Given the execution results of a *distinguishable program*, we analyzed the impact of each Wasm runtime on this anomaly respectively, based on the execution time of this program on each runtime. Specifically, for each dimension in the normalized execution time vector, we calculated its distance from the value of the same dimension in the normalized vector of the *oracle ratio*. This distance is called *deviation degree*, representing the impact of the corresponding Wasm runtime on this anomaly. Thus, we treated the Wasm runtime with the largest *deviation degree* as the issue-related runtime. Then, we analyzed the scenarios where these performance issues occurred based on the source code of the related test programs, and we verified the issues by reproducing them on similar test programs. Finally, we summarized seven performance issues in three Wasm runtimes, which are all previously unknown issues. The results are shown in Table III. Currently, all the issues are confirmed by developers, and four issues have been fixed.

For Wasmtime, we identified three performance issues. Issue #7731 occurred when Wasmtime handling floating-point (FP) arithmetic. The inserted operator in the issue-related test program is from a historical abnormal test program, which contains multiple FP arithmetic expressions. It is worth noting that the historical test program did not reveal this issue in the previous study. This verifies the effectiveness of the first design insight of *WarpGen*, *i.e.*, the historical issue-triggering test programs can help detect new issues. Issue #7732 is related to the improper optimization when accessing the pointers to constant. In this case, the inserted operator is from a newly generated test program, where the anomaly occurred when accessing a variable of type ‘`const int32_t *`’. This result also verifies the effectiveness of the *distinguishability*-guided design of *WarpGen*, *i.e.*, the newly extracted operators also help detect performance issues. Issue #7733 was caused by an increment operation (*i.e.*, `x++`) in nested loops.

For Wasmer, we also identified three performance issues. Issue #4378 occurred when Wasmer executed the operations on FP arrays, which was reflected in multiple test programs. Specifically, we found that abnormal latency occurred on the operations of arrays with types of `float[]`, `double[]`, and `double*[]`. Issue #4379 reflects the improper handling of Wasmer when calling the standard output function `fprintf()`. Issue #4380 was caused by insufficient optimization when accessing variable addresses.

For WAMR, we identified one performance issue (Issue #2938) related to insufficient FP arithmetic optimization, similar to Issue #7731 in Wasmtime. This issue was also reflected in multiple test programs, one with the same inserted operator as Issue #7731. This result indicates that the same inserted operator in different contexts (*i.e.*, seed programs) can trigger performance issues in different Wasm runtimes.

In summary, the results verify the ability of *WarpGen* to detect new performance issues in Wasm runtimes. The results also show the effectiveness of the key design insights (*i.e.*, history-driven and *distinguishability*-guided) in *WarpGen*, which indicates that *WarpGen* is a practical test program generation approach for Wasm runtime performance testing.

V. DISCUSSION

A. Case Study

In this subsection, we provide a case study to show the effect of *WarpGen* intuitively. Due to space limitations, we just show one representative case and leave others in issue reports.

```
for (i = 0; i < 20; i++)
{
  transparent_crc_bytes(&g_16[i], sizeof(g_16[i]), "g_16[i]", print_hash_value);
  if (print_hash_value) printf("index = [%d]\n", i);
  ...
  // inserted code
  ...
  g_1086 = (g_499+g_14+g_499)/(g_1875+g_499+g_14+g_499);
  g_1875 = g_499;
  g_1086 = x * ( g_1086 + g_1875 + g_499 * s ) / g_499;
  g_1875 = g_1875 / g_1086;
  n  = (long)( (double)( 40000 * (long)g_1875 ) / g_1875 );
  g_14 = g_1875 - 25.2;
  ...
}
```

Fig. 5. Test program related to Issue #7731.

Figure 5 shows part of the code in the test program related to Issue #7731. The inserted operator in this test program was extracted from a historical abnormal case `flops-1.c` reported by *WarpDiff*. It contains several FP arithmetic expressions, where the original variables were all replaced with same-type variables in the seed program (*i.e.*, variables whose names begin with ‘`g_`’, type of `double`). This operator was inserted in a loop (`for` statement) of the seed program. When this test program was executed, Wasmtime experienced severe abnormal latency, which was about three times slower than its expected performance. However, this issue has not been discovered in the previous study. We found that this issue cannot be triggered when Wasmtime executes `flops-1.c` alone. It only occurred in the specific context of this generated test program. We further investigated the reason and found that the inserted code was executed in a loop, amplifying the abnormal performance. Meanwhile, the modified variables changed the original logic of the seed program, which also contributed to this anomaly.

This case study indicates that performance issues in Wasm runtimes may be triggered under various unexpected conditions, which cannot be detected by existing test programs. *WarpGen* can produce diverse issue-triggering test programs to detect more performance issues in Wasm times with high efficiency. Therefore, *WarpGen* is a promising tool for Wasm runtime performance testing.

B. Threats to Validity

The threats to *internal* validity mainly lie in the implementation of *WarpGen*. To reduce the threats, we carefully checked all the code and conducted sufficient testing. For the Clang tools of operator extraction and program synthesis, we used large-scale C programs to verify their correctness. For the Java framework of *WarpGen*, we conducted multiple unit and integration tests for each module. We also implemented a comprehensive log system to monitor its workflow.

The threats to *external* validity mainly lie in the programs used for operator extraction and as the seeds. *WarpGen* extracted the initial operators from the abnormal test programs reported by *WarpDiff* since they are the only source of historical issue-triggering test programs we know. For the seed programs, we selected some random C programs generated by Csmith [20] in our experiments. Csmith is one of the most widely used C program generators, so it can provide a representative result.

The threats to *construct* validity mainly lie in the settings of parameters in the experiments. First, the parameter N affects the threshold for updating the *distinguishable programs* in *WarpGen*. We set N to 20 in our experiments based on the experience of *WarpDiff*, since the anomalies reflected in subsequent test programs are less significant. Second, the parameter M affects the speed of removing an operator. Based on the observations in multiple pre-experiments, we set M to 5 since it can achieve a trade-off between iteration efficiency and test program quality.

VI. Related Work

Wasm Runtime Performance. Wasm [1] is a binary instruction format serving as a compilation target for programming languages [2]–[6]. Wasm's fast, safe, lightweight, and portable features make it popular on both web side [37]–[39] and server side [40]–[43]. High performance is a critical design goal of Wasm. Regarding Wasm runtime performance, existing studies mainly focus on the web side [31]–[33], [44]–[46]. For example, Jangda *et al.* [31] designed BROWSIX-Wasm to conduct the first large-scale evaluation of the performance of Wasm vs. native. Wang [32] investigated how browser engines optimize Wasm execution in comparison to JavaScript. Yan *et al.* [33] further extended this study to three major browser engines (Chrome, Firefox, and Edge). Romano and Wang [46] recently investigated the counterintuitive impacts of function inlining on Wasm runtime performance on the web.

There are also a few studies on server-side Wasm runtime performance. Spies and Mock [34] evaluated Wasm runtime performance in non-web environments. Their measurements demonstrated that Wasm is generally faster than JavaScript. Recently, Jiang *et al.* [17] identified the significant impact of performance issues on server-side Wasm runtimes, and they proposed a differential testing approach *WarpDiff* to reveal performance issues in server-side Wasm runtimes. Some other studies [47]–[50] investigated Wasm runtime bugs and showed several cases related to runtime performance. However, state-of-the-art research on Wasm runtime performance testing still suffers from insufficient high-quality test programs, which we aim to solve in this work.

Test Program Generation. High-quality test programs are critical for software testing. Test program generation aims to construct effective and diverse test programs, typically applied to compiler testing [21], [51]–[56] and JVM testing [23], [57]–[60]. Existing program generation approaches can be generally divided into two categories: generation-based approaches and mutation-based approaches. Csmith [20] and YARPGen [61] are two typical program generators designed for generating C programs free of undefined behaviors. Alipour *et al.* [62] proposed directed swarm testing [63] that uses statistics and a variation of random testing to produce random tests. HiCOND [21] and K-Config [22] extract insights from historical bug reports and use the insights to guide the test program generators. Typical mutation-based approaches include *equivalence modulo inputs* (EMI) [64], Athena [65], and Hermes [66], which produce program variants by mutating code snippets in existing programs. CLsmith [67] follows the idea of EMI to validate OpenCL compilers by mutating dead code in test programs. Donaldson and Lascu [68] further proposed strategies for metamorphic testing of OpenGL compilers using opaque value injection. There are also some machine learning-based program generation approaches, *e.g.*, DeepSmith [69] and DeepFuzz [70].

Regarding Wasm runtime performance testing, a significant challenge is the lack of sufficient test programs. Although some studies aimed to generate Wasm code [71]–[74], none are targeted at runtime performance testing. So, in this work, we propose the first test program generation approach for Wasm runtime performance testing, improving the efficiency of discovering more performance issues in Wasm runtimes.

VII. Conclusion

In this paper, we propose *WarpGen*, a novel test program generation approach for Wasm runtime performance testing. It first extracts code snippets from historical issue-triggering test programs as operators, then inserts an operator into a random seed to generate a new test program. It adopts an indicator *distinguishability* to verify the quality of test programs and guide the program generation process. Our experiments have shown that *WarpGen* can generate high-quality test programs more efficiently than other baseline approaches. *WarpGen* has identified seven previously unknown performance issues in three Wasm runtimes. The results verify the effectiveness of *WarpGen* for Wasm runtime performance testing.

Acknowledgment

This work was supported by the Natural Science Foundation of Shanghai (No. 22ZR1407900) and the RGC Grant for Theme-based Research Scheme Project (RGC Ref. No. T43-513/23-N).

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] D. Lehmann, M. Thalakottur, F. Tip, and M. Pradel, "That'sa tough call: Studying the challenges of call graph construction for webassembly," in *Symposium on Software Testing and Analysis (ISSTA'23)*, 2023.

[3] D. Lehmann and M. Pradel, "Finding the dwarf: recovering precise types from webassembly binaries," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 410–425.

[4] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of {WebAssembly}," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 217–234.

[5] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 1045–1058.

[6] A. Romano and W. Wang, "Automated webassembly function purpose identification with semantics-aware analysis," in *Proceedings of the ACM Web Conference 2023*, 2023, pp. 2885–2894.

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

[8] 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.

[9] 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.

[10] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 265–279.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] "Wasmedge," https://github.com/WasmEdge/WasmEdge, 2024.

[17] S. Jiang, R. Zeng, Z. Rao, J. Gu, Y. Zhou, and M. R. Lyu, "Revealing performance issues in server-side webassembly runtimes via differential testing," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 661–672.

[18] "Secure & lightweight microservice with a database backend," https://github.com/second-state/microservice-rust-mysql, 2024.

[19] "Llvm test suite," https://github.com/llvm/llvm-test-suite, 2024.

[20] 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.

[21] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 305–316.

[22] M. R. I. Rabin and M. A. Alipour, "Configuring test generators using bug reports: a case study of gcc compiler and csmith," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1750–1758.

[23] 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.

[24] "Wasmer," https://github.com/wasmerio/wasmer, 2024.

[25] "Wasmtime," https://github.com/bytecodealliance/wasmtime, 2024.

[26] "Webassembly micro runtime (wamr)," https://github.com/bytecodealliance/wasm-micro-runtime, 2024.

[27] "V8 javascript engine," https://v8.dev/, 2024.

[28] "Spidermonkey javascript/webassembly engine," https://spidermonkey.dev/, 2024.

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

[30] "Awesome webassembly runtimes," https://github.com/appcypher/awesome-wasm-runtimes, 2024.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] "Llvm project," https://github.com/llvm/llvm-project, 2024.

[36] 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.

[37] 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.

[38] 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.

[39] A. Romano, D. Lehmann, M. Pradel, and W. Wang, "Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1574–1589.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] ——, "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.

[46] A. Romano and W. Wang, "When function inlining meets webassembly: Counterintuitive impacts on runtime performance," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2023.

[47] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A comprehensive study of webassembly runtime bugs," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 355–366.

[48] Y. Zhang, S. Cao, H. Wang, Z. Chen, X. Luo, D. Mu, Y. Ma, G. Huang, and X. Liu, "Characterizing and detecting webassembly runtime bugs," *arXiv preprint arXiv:2301.12102*, 2023.

[49] Z. Liu, D. Xiao, Z. Li, S. Wang, and W. Meng, "Exploring missed optimizations in webassembly optimizers," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 436–448.

[50] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 42–54.

[51] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," *Information and software technology*, vol. 39, no. 9, pp. 617–625, 1997.

[52] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 700–711.

[53] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 223–234.

[54] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[55] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 56–68.

[56] H. Tu, H. Jiang, X. Li, Z. Ren, Z. Zhou, and L. Jiang, "Remgen: Remanufacturing a random program generator for compiler testing," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 529–540.

[57] 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.

[58] 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.

[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] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, "Vectorizing program ingredients for better jvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 526–537.

[61] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[62] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 70–81.

[63] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 78–88.

[64] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 216–226.

[65] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 386–399.

[66] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications*, 2016, pp. 849–863.

[67] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 65–76.

[68] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st international workshop on metamorphic testing*, 2016, pp. 44–47.

[69] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[70] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1044–1051.

[71] K. Haßler and D. Maier, "Wafl: Binary-only webassembly fuzzing with fast snapshots," in *Reversing and Offensive-oriented Trends Symposium*, 2021, pp. 23–30.

[72] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly," *arXiv preprint arXiv:2110.15433*, 2021.

[73] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "Wadiff: A differential testing framework for webassembly runtimes," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 939–950.

[74] W. Zhao, R. Zeng, and Y. Zhou, "Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024, pp. 1035–1047.