Random Testing of the WebAssembly System Interface

Ethan Stanley
University of Utah

UUCS-24-003

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

20 April 2024

Abstract

The WebAssembly System Interface (WASI) enables WebAssembly (Wasm) programs to interact with the facilities of the computer on which the Wasm program runs. This greatly expands Wasm's utility and power outside of the browser, but greater capability increases the risk of vulnerabilities that arise when Wasm runtimes are not implemented correctly. This is especially important given Wasm's emphasis on securty. It is therefore necessary to thoroughly test implementations of WASI. We evaluate the effectiveness of random testing for finding bugs in implementations of WASI. We create a system to perform differential testing on Wasm runtimes. This system randomly generates Rust test cases that invoke system calls and compiles them to x86 assembly and Wasm. If the runtime behavior of these executables differ, we expect that there is a bug in one of the systems under test (SUT). We evaluate the effectiveness of our random testing framework by analyzing the bugs it discovers inWASI-compliant runtimes.

RANDOM TESTING OF THE WEBASSEMBLY SYSTEM INTERFACE

by

Ethan Stanley

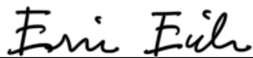
A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

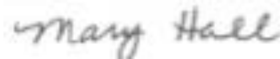
In

Computer Science

Approved:



Eric Eide
Thesis Faculty Supervisor



Mary Hall
Director, Kahlert School of Computing

Thomas C. Henderson
Honors Faculty Advisor

Monisha Pasupathi, PhD
Dean, Honors College

April 2024

Copyright © Ethan Stanley 2024
All Rights Reserved

ABSTRACT

The WebAssembly System Interface (WASI) enables WebAssembly (Wasm) programs to interact with the facilities of the computer on which the Wasm program runs. This greatly expands Wasm's utility and power outside of the browser, but greater capability increases the risk of vulnerabilities that arise when Wasm runtimes are not implemented correctly. This is especially important given Wasm's emphasis on security. It is therefore necessary to thoroughly test implementations of WASI. We evaluate the effectiveness of random testing for finding bugs in implementations of WASI. We create a system to perform differential testing on Wasm runtimes. This system randomly generates Rust test cases that invoke system calls and compiles them to x86 assembly and Wasm. If the runtime behavior of these executables differ, we expect that there is a bug in one of the systems under test (SUT). We evaluate the effectiveness of our random testing framework by analyzing the bugs it discovers in WASI-compliant runtimes.

CONTENTS

ABSTRACT . . . . . **ii**

LIST OF FIGURES . . . . . **iv**

CHAPTERS

1. INTRODUCTION **1**

2. BACKGROUND **3**

3. METHODS **5**

3.1 Test Case Generation 5
3.2 Test Harness 8
3.3 Bug Detection 8
3.4 Test Case Reduction 9

4. EVALUATION **10**

4.1 Results 10
4.2 Discussion 11

5. RELATED WORK **13**

6. CONCLUSIONS **15**

REFERENCES **17**

LIST OF FIGURES

3.1 Differential testing pipeline 6
3.2 Example of AST generated by Wasimilar 6

CHAPTER 1

INTRODUCTION

In this study, we evaluate the effectiveness of random testing for identifying bugs in WASI-compliant runtimes. Specifically, we use differential testing to identify randomly generated test cases that trigger bugs.

WebAssembly, or Wasm, is a portable compilation target that has been widely adopted in web browsers [17]. Recently, interest in using Wasm outside of the browser led to the creation of WASI. WASI is a system interface for Wasm that enables interaction with the operating system. It serves a similar purpose as C's standard library.

Wasm prioritizes security by employing a memory-safe, sandboxed execution environment [25]. WASI attempts to uphold Wasm's security guarantees by sandboxing I/O operations. Prior to execution, a Wasm program must be given explicit permission to use a feature of WASI. For example, WASI programs can only modify directories that are specified from the command line.

An incorrect implementation of WASI could undermine the security of Wasm. In addition, WASI greatly increases the capability of Wasm programs, so the consequences of vulnerabilities caused by bugs in WASI can be more severe. It is therefore necessary to thoroughly and effectively test WASI-compliant runtimes. At the time of writing, no efforts to apply random testing to WASI have been documented.

We achieve this by creating a differential testing framework for WASI. Testing WASI is a distinct goal from testing Wasm. To test Wasm, one would generate Wasm programs that maximize coverage of Wasm's syntax and semantics. In contrast, effectively testing WASI involves generating interesting and varied calls to the WASI API. This does not necessarily require all of the features of Wasm.

Differential testing is a software testing technique that attempts to detect bugs by providing the same input to different implementations of a software system and comparing

their behaviors [14]. If the specification of the systems allows only a single possible outcome, and the behaviors of the systems differ, then at least one of them must be inconsistent with its specification. Differential testing offers a solution to the test oracle problem: the challenge of distinguishing correct and incorrect behavior in software [1]. With differential testing, any test case can be fed to a software system as long as it does not contain unspecified or non-deterministic behavior. This combines powerfully with a random test case generator, for which determining correct behavior would otherwise be difficult.

In this study, we compare the behaviors of a Wasm program and an x86 binary. To get executables that we expect to behave equivalently, we take advantage of the fact that they are both compilation targets of Rust. Thus, given a Rust test case that invokes system calls, a pair of executables can be generated that is suitable for performing differential testing on WASI.

In the current version of WASI [6], API calls are provided for generating random numbers and interacting with the file system, clocks, and sockets. Our test case generator produces programs that maximize coverage of the file system, clock, and random APIs.

We hypothesize that random testing is an effective method for testing implementations of WASI. We evaluate this claim by analyzing the kinds of bugs discovered by our differential testing framework.

With our differential testing framework, we were able to find bugs in multiple implementations of WebAssembly.

WASI, like WebAssembly, is an evolving standard. Our differential testing framework can be used as a development tool for Wasm runtimes as WASI changes and becomes more widely implemented. Over the course of this project, many WebAssembly runtimes were updated to support WASI. We expect this trend to continue.

CHAPTER 2

BACKGROUND

Differential testing is widely used to find bugs in software systems. Differential testing can be applied to compilers using test cases from random program generators. Csmith is a well-known example of a test case generator that was used to perform differential testing on C compilers [26].

Creating a random program generator can be a complex process. For instance, the development of Csmith required hundreds of hours and nearly 40,000 lines of code [10]. There is particular difficulty in generating programs that are suitable as test cases for differential testing. These programs must avoid unspecified and non-deterministic behavior. Otherwise, one cannot be confident that differences in behavior between systems under test are due to the existence of a bug.

Recently, Xsmith was introduced to make the development of random program generators, especially those intended for differential testing, simpler [10]. Xsmith is a domain-specific language built within Racket [9] for defining random program generators. It allows the user to specify a grammar that is used to randomly construct an abstract syntax tree (AST) representing a program. Xsmith also allows the user to define choice methods and attributes to arbitrarily guide the generation of the AST. Xsmith contains predefined properties to make adding common programming language features easier.

Xsmith has been used to implement several random program generators that have been used to identify bugs in compilers [11, 24]. One of these program generators is Wasmlike [24], a generator of random Wasm programs. Wasmlike is an inspiration for this project, and one method of testing WASI implementations would be to extend Wasmlike to make use of the WASI API. This would result in the direct generation of WASI test cases, rather than obtaining them by compiling Rust test cases.

Instead, we chose to obtain the test cases indirectly so that our test case generator can

still be used as WASI evolves. We test targets that implement WASI Preview 1 [6], but more versions of WASI are in development. If future versions of WASI are supported by the Rust compiler, our test case generator can be used to test them as well. Another reason we chose to obtain test cases indirectly was to have multiple targets for differential testing. When we began development of our test case generator, we were only aware of one reliable implementation of WASI. Therefore, it would have been impossible to perform differential testing if our generator produced Wasm programs. By the end of our project, we had selected five implementations of WASI as targets.

CHAPTER 3

METHODS

We built a framework for performing differential testing on WASI runtimes. Figure 3.1 illustrates our differential testing pipeline. A test case is first obtained from our test case generator. The test case is subsequently compiled to Wasm and x86 assembly. The Wasm program is given to one or more WASI-compliant runtimes. The x86 binary is executed natively. The results of each execution of the test case are compared. If there is divergent behavior, the test case is logged as potentially bug-triggering. This process repeats for the duration of a testing campaign. Additional work is needed to reduce and verify test cases that are marked as bug triggering. More detail about each of these steps is provided in the following sections.

3.1 Test Case Generation

To obtain Rust test cases, we built a test case generator called Wasimilar. We created it using Xsmith. Xsmith allows the user to define a grammar and arbitrary functions to guide generation of random abstract syntax trees (ASTs) representing programs. Typically, the user defines a grammar that mirrors the grammar specification of a programming language. Instead, we generate relatively flat ASTs because our goal is to generate interesting calls to the WASI API. This does not require all of Rust's semantics and syntax. The ASTs we generate consist of a single top-level ″program″ node with a configurable number of children, each representing an interaction with the WASI API. An example of an AST generated by Wasimilar is shown in Figure 3.2.

Wasimilar generates Rust programs that contain random sequences of operations supported by WASI. Every generated operation is wrapped inside a node. The Rust code generated for a node performs any necessary setup for the operation and checks that the operation will succeed given the current state of execution. For example, the code generated for a file-write node checks if there are any file descriptors that can be written

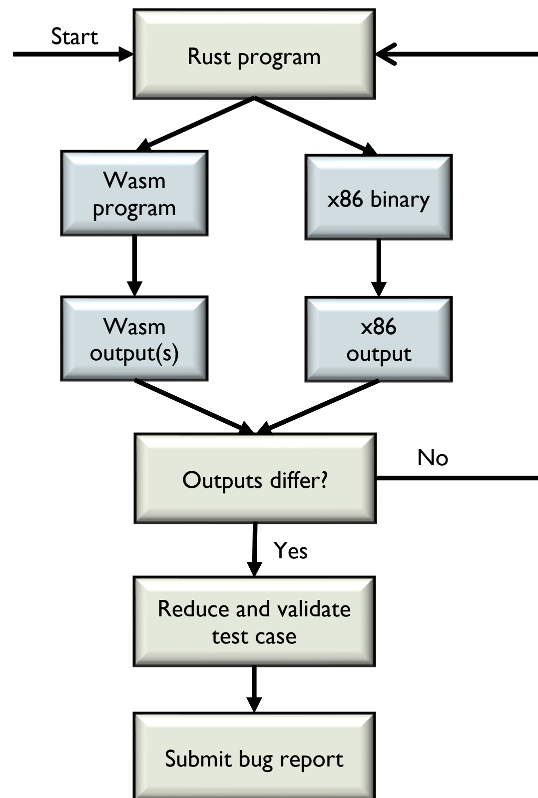


Figure 3.1. Differential testing pipeline

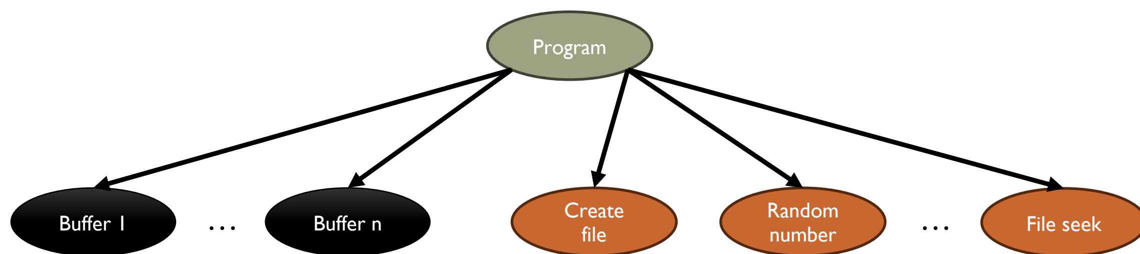


Figure 3.2. Example of AST generated by Wasimilar

to and randomly selects one if so. If there are none, no file-write occurs and execution continues with the next node. As a result, there are no dependencies pertaining to the order of generated nodes.

Although packaging operations into nodes ensures that they can be generated in any order, operations can still affect one another. One way they do this is through shared buffers. Operations can read from and write to buffers. In this way, operations without visible effects, such as requests for random numbers, can be made visible by other operations, such as a file-write. Below is a snippet of a generated test case. It demonstrates two operations, a file-read and a file-write, and their containing nodes. Through a shared buffer, *lift_1*, the result of the file-read affects the input to the file-write.

```
// shared buffer
let mut lift_1 = String::from("mpvlprcxqsb...");

// file read node
if open_file_ptrs.len() > 0 {
    let mut fp = open_file_ptrs.choose(&mut rng).unwrap();
    fp.read_to_string(&mut lift_1).unwrap();
}

// file write node
if open_file_ptrs.len() > 0 {
    let index: usize = rng
        .gen_range(0..open_file_ptrs.len() as i32)
        .try_into()
        .unwrap();
    write!(open_file_ptrs[index], "{}", lift_1.to_string()).unwrap();
}
```

This code snippet demonstrates another property of the test case generator: file operations select their file operands at run time. In contrast to buffers that are assigned to operations during generation, operations dynamically select a file to operate on. Such operations include writing to a file, reading from a file, closing a file, reopening a file, and deleting a file. In addition, the locations of newly created files and directories are determined randomly at run time. All random choices are made with a seeded random number generator in order to avoid non-deterministic behavior.

An AST is turned into a test case by placing the code snippets of each node sequentially

in the main function of a Rust program. At the start of the main function, a seeded random number generator and data structures to store the state of the file system are initialized.

Wasimilar generates test cases that contain operations with the filesystem, random number generation, and clocks. Although WASI Preview 1 supports some operations with sockets, there is no support for creating a socket and connecting it to a server. In addition, workarounds exist only for some implementations of WASI and do not have a uniform interface. Thus, our test cases do not contain operations with sockets.

3.2 Test Harness

We created a configurable test harness to automate the process of differential testing. The harness first queries Wasimilar for a test case. Next, it compiles the test case to Wasm and x86 assembly. The x86 binary is executed natively, while the Wasm binary is given to the WASI runtimes. Each of these execution environments is a test target. Each target executes its test case in an empty directory. The standard output of each target is captured in a special log file. A predicate script is used to detect any difference in behavior of the targets. If a difference is detected, the seed of the test case is recorded. The harness proceeds to clear the effects of executing the test case and repeats this process until a specified number of test cases have been evaluated.

We carried out testing campaigns on CloudLab [8], a cloud computing testbed for research. This allowed us to run extended campaigns and have our test harness running on multiple machines. This helped us evaluate a large number of test cases.

3.3 Bug Detection

The predicate script compares the results of each WASI runtime with the results of running the x86 binary natively. First, it compares all the files and subdirectories created by the test case, including the logged standard output, to ensure they are equivalent. Next, it makes a special comparison for a file containing all requests made for the current time. Since this is inherently a non-deterministic operation, direct comparison will fail. Instead, a regular expression is used to count the number of times the current time was successfully queried. This quantity is compared instead. If no difference is detected through either comparison, the test case is discarded because no divergent behavior (arising from a bug

in a target) has been detected.

3.4 Test Case Reduction

When a test case is determined to cause different behavior among the targets, further work is needed to determine if it is bug-revealing. We use C-Reduce, a tool for simplifying C programs while preserving interesting behavior [16]. C-Reduce has been found to work well for test cases in other languages, including Rust. After automated reduction with a tool, it is often necessary to further reduce the test case by hand. If we reduce a test case and determine it does not contain non-deterministic or unspecified behavior, we can conclude that a bug has been found and submit the reduced test case in a bug report.

CHAPTER 4

EVALUATION

To get an accurate assessment of the effectiveness of our random testing framework, we ran large scale testing campaigns. This was facilitated by CloudLab, a cloud computing testbed for research. CloudLab allows users to reserve machines for various purposes. We allocated one to three machines and installed our testing harness and test case generator on each of them. We ran testing campaigns on these machines ranging from 1 to 5 days in duration. At the end of a testing campaign, we collected the list of seeds that yield test cases exhibiting divergent behavior when input to our random program generator. We tested the following WASI-compliant runtimes in our testing campaigns: Wasmtime version 18.0.1 [2], Wasmer version 4.2.5 [23], WasmEdge version 0.13.5 [5], WAMR version 1.3.2 [4], and Wasmi version 0.31.2 [15].

Each machine carried out a testing campaign independently from the others. Therefore, it is possible that some seeds were redundantly evaluated by multiple machines.

4.1 Results

Our testing campaigns identified two bugs in implementations of WASI. The first was a bug in the WasmEdge runtime [5]. The Rust test case, shown below, is simple and attempts to create a new directory.

```
use std::fs;

fn main() {
    fs::create_dir("foo/").unwrap();
}
```

The bug is a result of the trailing slash in the name of the new directory. WasmEdge raised an exception, which is inconsistent with the specification. This bug was confirmed and fixed by the developers [20].

The second bug was found in the Wasmer runtime [23]. Its test case is shown below.

```
use std::fs::OpenOptions;
use std::io::prelude::*;
use std::io::SeekFrom;

fn main() {
    let mut fp = OpenOptions::new()
        .append(true)
        .read(true)
        .create(true)
        .open("file")
        .unwrap();

    write!(fp, "{}", "a").unwrap();
    let _ = fp.rewind();
    write!(fp, "{}", "b").unwrap();
    println!("{}", fp.seek(SeekFrom::Current(0)).unwrap());
}
```

This test case opens a file in append mode and subsequently performs write and seek operations on it. Wasmer has an incorrect file offset after the second write operation, and this is made visible with the print statement. This bug was confirmed by the developers but has not been fixed at the time of writing [21].

A summary of these bugs can be found in the table below.

System	Link to report	Fixed
WasmEdge v0.13.5	https://github.com/WasmEdge/WasmEdge/issues/3231	Yes
Wasmer v4.2.5	https://github.com/wasmerio/wasmer/issues/4469	No

4.2 Discussion

Both of the bugs identified by our random testing framework were related to WASI. Therefore, we have evidence that our testing framework is capable of finding WASI-related bugs in Wasm runtimes.

Our test case generator involves systems other than the subset of each target implementing WASI. For example, our test cases are compiled using the Rust compiler. They are given to runtimes that implement the syntax and semantics of Wasm. We also execute binaries natively. Thus, it is possible that a bug-revealing test case might not be related to

WASI. This makes the task of verifying bug-revealing test cases more challenging.

This issue arose when analyzing the test case shown below, which was flagged by our testing harness for exhibiting divergent behavior.

```
use rand::{Rng,SeedableRng};
use rand::rngs::StdRng;

fn main() -> std::io::Result<()>{
    let mut rng = StdRng::seed_from_u64(568596);
    let open_file_ptrs = vec![0];
    rng.gen_range(0..open_file_ptrs.len());
    println!("{}", rng.gen_range(0..2));
    Ok(())
}
```

This test case prints a different random value when compiled to x86 and Wasm. Ultimately, this was due to platform-specific behavior in the Rust specification, not a bug in an implementation of WASI. The size of the *usize* type, which is returned by *open_file_ptrs.len()*, is dependent on the compilation target [18]. This caused the pseudo-random number generator (PRNG) to consume a different number of bits across different targets, because the representation of the range passed to *rng.gen_range()* was different across those targets. This affected subsequent queries to the PRNG, which was detected as divergent behavior. This is an example of the more general problem of avoiding platform-specific, unspecified, and non-deterministic behavior in randomly generated test cases, exacerbated by the fact that our test cases adhere to multiple specifications.

CHAPTER 5

RELATED WORK

The technique of compiling to x86 and Wasm from a high level language was previously used by Stiévenart et al. to evaluate the security risks of porting C programs to Wasm [22]. They evaluated the equivalence of the binaries by comparing their return codes and standard output. We use these same comparisons, but we extend them to include the effects on the file system.

A major component of this study is developing a random program generator to provide Rust test cases. Several Rust program generators already exist [19], [7]. Sharma et al. introduced RustSmith, a Rust language fuzzer for differential testing implemented with Kotlin. RustSmith was used to find historical and previously unreported bugs in various Rust compilers. Dewey et al. created a Rust test case generator to demonstrate how constraint logic programming can be used to generate well-typed programs [7]. With this tool, they tested the Rust typechecker by generating programs that either adhere to or subtly violate Rust's type rules. Our test case generator differs from both of these program generators because it prioritizes generation of interesting sequences of system calls over stressing a compiler's adherence to Rust's semantics.

To our knowledge, no prior attempts to perform random testing on implementations of WASI have been documented. However, they have been tested in other ways. Users of Wasmtime regularly submit bug reports related to WASI [3]. These reports were recently categorized by Zhang et al. when they developed a taxonomy of bugs discovered in Wasm implementations [27]. They created a testing framework by collecting known bug-revealing test cases into a test suite. This was effective at finding bugs in Wasm implementations. However, it is still beneficial to have a framework for generating an arbitrary number of test cases, rather than having to update a test suite by hand. Additionally, a random testing framework can reveal bugs before they are encountered by users.

WASI can be viewed as a runtime library for Wasm. Li et al. introduced PyRTFuzz, a random testing framework for detecting bugs in Python runtimes, including runtime libraries [13]. They develop a system to automatically extract API descriptions of runtime libraries and use the extracted API descriptions to make diverse and valid calls to the runtime APIs, ensuring that they are fully covered. Mutational techniques are used to generate inputs for test cases. Although our test cases do not accept any input, WASI provides support for command line parameters and reading from standard input. Thus, their strategy of using mutation to generate interesting inputs could enhance the bug-finding power of our random testing framework. Ultimately, developing a system to automatically extract API descriptions for WASI would not be useful, as it is a single, relatively simple, API. This contrasts with Python runtimes, which have a large and growing number of APIs.

Many of the WASI operations supported by our test case generator involve the file system. Thus, it is important for our generator to produce interesting and varied interactions with the file system. Kim et al. introduce Hydra as a solution for this problem [12]. Hydra is a framework for fuzzing file systems that works by mutating both the sequence of system calls and the image of the file system. Any code checker can be connected to Hydra to serve as a test oracle. We take a generative approach to fuzzing WASI rather than a mutational approach. We also do not pre-populate our file system before executing a test case. Thus, we have no file system image to mutate. Our test cases also contain operations with clocks and random numbers in addition to the file system.

CHAPTER 6

CONCLUSIONS

In summary, we developed a random testing framework and used it to find two bugs in implementations of WASI. Thus, we can conclude that random testing is effective for finding bugs in implementations of WASI. We also showed that it is possible to specifically test the subset of Wasm runtimes that implement the WASI API.

We found that Xsmith is a useful tool for generating test cases even if they are not generated according to the grammar of a language. Although our test cases are in Rust, our test case generator was minimally concerned with the syntax and semantics of Rust. We generated shallow and deterministically sized ASTs, which is another unusual use case for Xsmith. Despite this, our test case generator benefited from Xsmith's framework for randomly constructing an AST, pairing reference nodes with binding nodes, and rendering a test case from an AST.

We conclude that the results of this study are promising, and we consider several avenues for future work. First, we could increase the power of our current random testing framework. This could include incorporating more WASI-runtimes into our testing harness, increasing the number of API calls supported by our test case generator, or generating test cases with more sophisticated control flow, e.g., incorporating functions, loops, and more conditional expressions. Second, we could employ a different approach to randomly test WASI runtimes. For example, one could develop a grey-box fuzzer that mutates test cases based on coverage feedback from an instrumented runtime. Another approach is to directly generate Wasm programs that make use of the WASI API. This would allow more fine-grained testing of WASI.

We expect that random testing will continue to be an effective tool for testing WASI runtimes as the API evolves and becomes more complex. Differential testing becomes a more powerful technique as the number of WASI-compliant runtimes grows. We hope that

our random testing framework will be a useful tool for developers implementing WASI runtimes.

REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 05 (May 2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[2] The Bytecode Alliance. 2023. Wasmtime. https://github.com/bytecodealliance/wasmtime.

[3] The Bytecode Alliance. 2023. Wasmtime WASI Issues. https://github.com/bytecodealliance/wasmtime/issues.

[4] The Bytecode Alliance. 2024. WebAssembly Micro Runtime. https://github.com/bytecodealliance/wasm-micro-runtime.

[5] Cloud Native Computing Foundation. 2024. WasmEdge. https://github.com/WasmEdge/WasmEdge.

[6] Contributors to the WASI Specification. 2023. WASI Preview 1. https://github.com/WebAssembly/WASI/blob/main/legacy/preview1/docs.md.

[7] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust Typechecker Using CLP. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 482–493. https://doi.org/10.1109/ASE.2015.65

[8] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. https://www.flux.utah.edu/paper/duplyakin-atc19

[9] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

[10] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. 2023. Generating Conforming Programs with Xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Cascais, Portugal) *(GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 86–99. https://doi.org/10.1145/3624007.3624056

[11] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamarić, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 556–567. https://doi.org/10.1145/3533767.3534382

[12] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 147–161. https://doi.org/10.1145/3341301.3359662

[13] Wen Li, Haoran Yang, Xiapu Luo, Long Cheng, and Haipeng Cai. 2023. PyRTFuzz: Detecting Bugs in Python Runtimes via Two-Level Collaborative Fuzzing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (¡conf-loc¿, ¡city¿Copenhagen¡/city¿, ¡country¿Denmark¡/country¿, ¡/conf-loc¿) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1645–1659. https://doi.org/10.1145/3576915.3623166

[14] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://dblp.uni-trier.de/db/journals/dtj/dtj10.html#McKeeman98

[15] Parity Technologies. 2024. Wasmi. https://github.com/wasmi-labs/wasmi.

[16] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[17] Andreas Rossberg. 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/

[18] The Rust Foundation. 2024. Primitive Type usize. https://doc.rust-lang.org/std/primitive.usize.html.

[19] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. https://doi.org/10.1145/3597926.3604919

[20] Ethan Stanley. 2024. WasmEdge Issue 3231. https://github.com/WasmEdge/WasmEdge/issues/3231.

[21] Ethan Stanley. 2024. Wasmer Issue 4469. https://github.com/wasmerio/wasmer/issues/4469.

[22] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. 2022. Security Risks of Porting C Programs to WebAssembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22)*. Association for Computing Machinery, New York, NY, USA, 1713–1722. https://doi.org/10.1145/3477314.3507308

[23] Wasmer, Inc. 2024. Wasmer. https://github.com/wasmerio/wasmer.

[24] Guy Watson. 2023. *Random Testing of WebAssembly Implementations Using Semantically Valid Programs*. Master's thesis. University of Utah.

[25] WebAssembly Community Group. 2019. WebAssembly. https://webassembly.org/. Accessed: 2023-22-11.

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532

[27] Yixuan Zhang, Shangtong Cao, Haoyu Wang, Zhenpeng Chen, Xiapu Luo, Dongliang Mu, Yun Ma, Gang Huang, and Xuanzhe Liu. 2023. Characterizing and Detecting WebAssembly Runtime Bugs. *ACM Trans. Softw. Eng. Methodol.* (Sep 2023). https://doi.org/10.1145/3624743

Name of Candidate: Ethan Stanley

Date of Submission: April 19, 2024