[WIP] Differential testing WASI paper

Author One
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email@example.com

Author Two
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email@example.com

***Abstract*—Abstract here.**

I. Introduction

Introduction here.

II. Background and Motivation

Background and motivation here.

III. Methodology

WAZZI aims to semi-automate uncovering platform-specific WASI implementation bugs in Wasm runtimes. In Fig. 1 we present the workflow of WAZZI, which consists of three automated phases: (1) interface analysis, (2) call generation, and (3) differential execution. During the interface analysis phase, WAZZI parses an annotated WASI specification to establish the semantic constraints of each function. With these semantic constraints, the call generation phase selects an appropriate WASI function to call and prepares interesting inputs. Then, in differential execution phase, WAZZI executes the generated call, checks outputs and side effects, and minimizes the problematic call sequence if it finds a behavioral divergence. Starting with a motivating example, we elaborate on why WAZZI can overcome challenges in previous solutions. We then describe the overall workflow of WAZZI and its major components.

A. Motivating Example

Many real-world applications depend on filesystem services and implicitly rely on filesystem-related syscalls behaving as expected for correctness. Fig. 2a shows a simplified example of a C application relying the correction behavior of `lseek` on line 10 to ensure a later `pwrite` does not corrupt important data. Fig 2b shows the same example compiled to Wasm with WASI support. In this example, the behavior of `lseek(fd0, 0 , SEEK_CUR)`, and its corresponding WASI function `fd_tell` depends on not only the values of its arguments, but also hidden system state they represent. Specifically, `fd0` is just an integer file descriptor, but the resource it represents—an open file description—contains information passed to a previous `open` syscall on line 1; in this case, the file access mode `O_WRONLY` and the flag `O_APPEND`.

In a bug discovered by WAZZI, one Wasm runtime will, in the presence of the `O_APPEND` flag during opening a file, always return offset as 0 regardless of the actual offset. Such a behavior violates the contract between application developer and the OS. In this case, the errant behavior is a result of the Wasm runtime emulating parts of the filesystem layer and incorrectly handling the append flag.

In this example, a hypothetical differential fuzzer can spot the bug after the `lseek()` by generating a precise `open–write–lseek` call sequence. However, prior techniques lack the semantic awareness to generate such call sequences. For example, WADIFF [1] generates Wasm binaries targeting a single instruction, and thus can only uncover inconsistencies at the instruction level.

B. Overall Workflow

C. Interface Analysis

D. Call Generation

E. Differential Execution

References

[1] S. Zhou, M. Jiang, W. Chen, H. Zhou, H. Wang, and X. Luo, "WADIFF: A differential testing framework for webassembly runtimes," in *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 2023, pp. 939–950. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00188

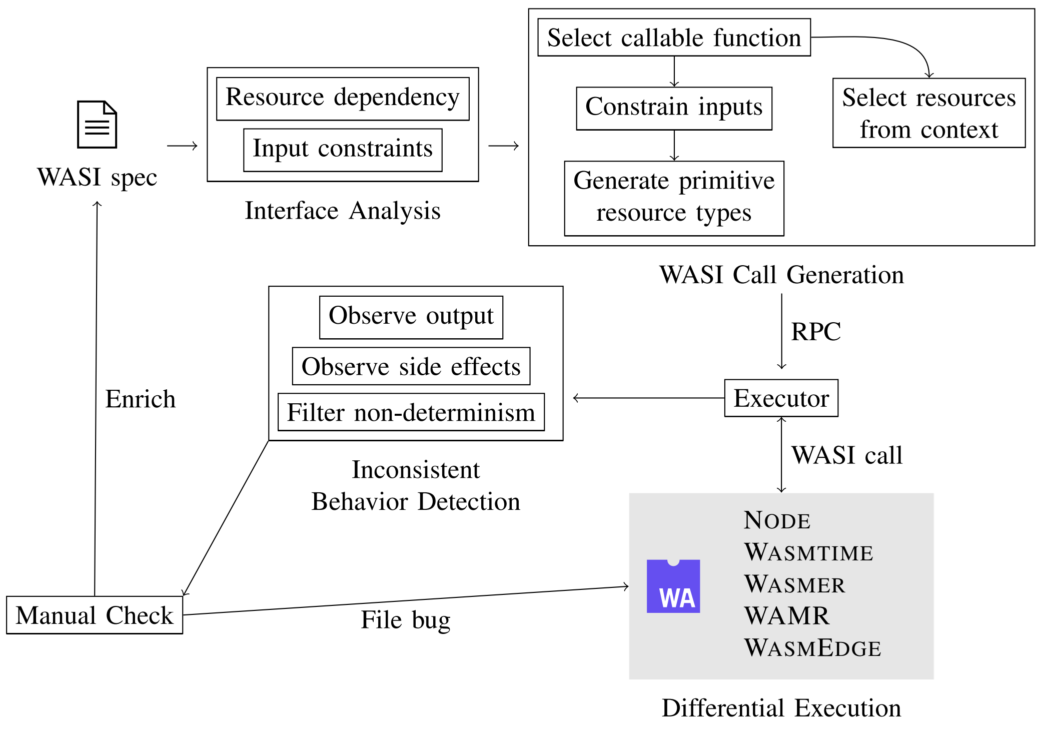

Fig. 1: The WAZZI differential testing workflow.

```
int fd0 = open(
    "file",
    O_WRONLY | O_APPEND
);
write(
    fd0,
    critical_data,
    len
);
off_t offset = lseek(
    fd0, 0, SEEK_CUR);
close(fd0);

int fd1 = open(
    "file",
    O_WRONLY
);
pwrite(
    fd1,
    data,
    data_len,
    offset
);
```

(a) Application code in C.

```

(call $path_open
  (; args omitted ;)
  (i32.const 0) (; fd0 ;))
(call $fd_write
  (i32.load 0) (; fd0 ;)
  (; omitted ;))
(call $fd_tell
  (i32.load 0) (; fd0 ;)
  (i32.const 4)
    (; offset ;))
(call $fd_close
  (i32.load 0) (; fd0 ;))

(call $path_open
  (; args omitted ;)
  (i32.const 0) (; fd1 ;))
(call $fd_pwrite
  (; args omitted ;)
  (i32.const 0) (; fd1 ;)
  (; omitted ;)
  (i64.load 4)
    (; offset ;))
```

(b) Compiled to WASI.

Fig. 2: Simplified example of application relying correct `lseek` behavior.