Execution-Aware Program Reduction for WebAssembly via Record and Replay

Doehyun Baek
doehyunbaek@gmail.com
University of Stuttgart
Stuttgart, Germany

Daniel Lehmann
mail@dlehmann.eu
Google Germany GmbH
Munich, Germany

Ben L. Titzer
btitzer@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, USA

Sukyoung Ryu
sryu.cs@kaist.ac.kr
KAIST
Daejeon, South Korea

Michael Pradel
michael@binaervarianz.de
CISPA Helmholtz Center
for Information Security
Stuttgart, Germany

***Abstract*—WebAssembly (Wasm) programs may trigger bugs in their engine implementations. To aid debugging, program reduction techniques try to produce a smaller variant of the input program that still triggers the bug. However, existing execution-unaware program reduction techniques struggle with large and complex Wasm programs, because they rely on static information and apply syntactic transformations, while ignoring the valuable information offered by the input program's execution behavior. We present RR-Reduce and Hybrid-Reduce, novel execution-aware program reduction techniques that leverage execution behaviors via record and replay. RR-Reduce identifies a bug-triggering function as the target function, isolates that function from the rest of the program, and generates a reduced program that replays only the interactions between the target function and the rest of the program. Hybrid-Reduce combines a complementary execution-unaware reduction technique with RR-Reduce to further reduce program size. We evaluate RR-Reduce and Hybrid-Reduce on 28 Wasm programs that trigger a diverse set of bugs in three engines. On average, RR-Reduce reduces the programs to 1.20% of their original size in 14.5 minutes, which outperforms the state of the art by 33.15× in terms of reduction time. Hybrid-Reduce reduces the programs to 0.13% of their original size in 3.5 hours, which outperforms the state of the art by 3.42× in terms of reduced program size and 2.26× in terms of reduction time. We envision RR-Reduce as the go-to tool for rapid, on-demand debugging in minutes, and Hybrid-Reduce for scenarios where developers require the smallest possible programs.**

I. Introduction

WebAssembly (Wasm) [1] is a language for the web designed for efficient, sandboxed execution. Wasm engines, like any other software, may contain bugs. Indeed, several techniques for detecting bugs in Wasm implementations have been proposed recently [2], [3], [4], [5]. These bug-triggering Wasm programs are often large and complex, which makes debugging the affected engine challenging.

Program reduction, which aims to find a smaller variant of the input program that still triggers the bug, is used to mitigate this problem. Delta Debugging [6] is the pioneering input reduction technique, and various others have been proposed since then [7], [8], [9]. For reducing Wasm programs, there are two industrially supported tools: wasm-reduce [10] and wasm-shrink [11]. While these tools are effective in some cases, they are limited in terms of both effectiveness and efficiency.

To illustrate the limitations of currently available techniques, consider the *commanderkeen* program [12], which reveals a miscompilation bug in the Wizard engine [13]. The Wasm binary is 3.9MB in size, which is not uncommon for binaries compiled from large programs. Wasm-reduce, i.e., the most effective existing tool for Wasm, reduces commanderkeen to 1.3MB after 24 hours of processing, which is far too large for a human to manually debug. Wasm-reduce is an example of an execution-unaware program reduction technique, i.e., it ignores the execution behavior of the input program. We identify two reasons why such execution-unaware techniques struggle with our example program and other input programs: (1) The reduction process applies syntactic transformations based on static information only, leading to a huge search space of possible reduced programs. (2) Checking whether the bug is preserved by a reduced program is expensive, as the Wasm engine has to execute the program to check whether the bug is still triggered. For the commanderkeen program, each such oracle invocation takes about 9 seconds, which is a long time for a program reduction technique that needs to perform many oracle invocations to find a reduced program.

A detailed inspection of the above example and other bug-triggering Wasm programs reveals an interesting observation: Usually, a single function in the program is responsible for triggering the bug. For example, the commanderkeen program contains 1,970 functions, but only one of them is relevant for triggering the bug. A naive approach to reduce the program would be to simply remove all other functions from the Wasm binary. However, this naive approach would lead to a program that fails to compile and execute, hence failing to trigger the bug, as the bug-triggering function depends on other functions and other code in the program. For the motivating example, 580 other functions are executed before the bug is triggered.

```
;; Function 1 is the bug-triggering function.
;; When compiling call_indirect instruction under specific state,
;; a register allocation bug in a JIT compiler is triggered.
(module
  (func (;0;) ;; entry to the bug-triggering function 1)
  (func (;1;)
    ;; omitted for brevity
    (call_indirect (i32.const 0) (i32.const 0) (local.get 0))
    ;; omitted for brevity)
  (func (;2;) (param i32 i32) (nop))
  (func (;3;) ;; update state for the bug to be triggered)
  (func (;4;) (param i32 i32) (result i32) (i32.const 0)))
```

Fig. 1: Hybrid-Reduce's output for commanderkeen.

We hypothesize that these problems can be addressed by leveraging the execution behavior of the input program, which provides valuable information about the dependencies between different parts of the program. This information may be used for isolating the bug-triggering function and any other code necessary to trigger a bug-exposing execution. Based on this hypothesis, we ask the question: Can we use information about the execution of an input program to reduce the input program more effectively?

To answer this question, this paper presents RR-Reduce, an *execution-aware* program reduction technique for Wasm. A key insight of the approach is that we can repurpose existing record and replay techniques for execution-aware program reduction. Record and replay is a debugging technique that records a program execution and then replays the recorded execution. It has been adopted in many domains, including native binaries [14], [15], JavaScript [16], [17], and Wasm [12]. Usually, record and replay is used to provide a deterministic way to replay an entire execution of a program. Instead, we use this technique to replay only parts of an input program that are essential to trigger the bug.

More specifically, RR-Reduce performs three steps: First, identify a bug-triggering function of the input program, which we call the target function. Second, partition the input program into two subprograms: one containing the target function and another containing the remaining functions. Third, use record and replay to generate a new program that includes the unmodified target function and generated replay functions that mimic the behavior of the remaining functions. Our RR-Reduce approach can be used as a stand-alone program reduction technique, or in combination with existing program reduction techniques. We call the latter approach Hybrid-Reduce, which applies an existing reduction technique (wasm-reduce) to the output of RR-Reduce. This further reduces output, at the expense of additional time.

Getting back to our motivating example, Figure 1 shows the reduced program produced by Hybrid-Reduce. Instead of the 1.3MB output of the existing wasm-reduce tool, our approach yields a reduced program of only 158 bytes. Such a small program enables engine developers to debug the bug in a reasonable time, ultimately leading to more robust implementations of Wasm.

We evaluate RR-Reduce and Hybrid-Reduce on 28 Wasm programs that trigger bugs in three Wasm engines. The results show that Hybrid-Reduce clearly outperforms the state of the art, wasm-reduce, and that our two approaches offer different effectiveness-efficiency trade-offs. Specifically, RR-Reduce reduces the amount of code a developer must inspect to 1.20% of the original size, on average, while taking 871 seconds, which is a 33.15x improvement in efficiency over the state of the art. That is, RR-Reduce significantly reduces input programs while offering unprecedented efficiency. The Hybrid-Reduce approach further reduces the program to 0.13% of its original size, while taking about 3.5 hours, on average. With these results, Hybrid-Reduce outperforms the state of the art both in terms of the resulting program size (3.42× improvement) and in terms of reduction time (2.26× improvement).

$$
\begin{array}{rcl}
\textit{module} & ::= & \textit{function}^* \ \textit{global}^* \ \textit{table}^* \ \textit{memory}^* \\
\textit{function} & ::= & \textit{type}_{\textit{func}} \ (\textit{import} \mid \textit{code}) \ \textit{export}^* \\
\textit{global} & ::= & \textit{type}_{\textit{val}} \ (\textit{import} \mid \textit{init}) \ \textit{export}^* \\
\textit{table} & ::= & \textit{import}^? \ \textit{idx}_{\textit{func}}{}^* \ \textit{export}^* \\
\textit{memory} & ::= & \textit{import}^? \ \texttt{byte}^* \ \textit{export}^* \\
\textit{code} & ::= & (\textbf{local} \ \textit{type}_{\textit{val}})^* \ \textit{instr}^* \\
\textit{init} & ::= & \textit{instr}^* \\
\textit{instr} & ::= & \textit{type}_{\textit{val}}\textbf{.const} \ \textit{value} \mid \textit{type}_{\textit{val}}\textbf{.load} \\
 & \mid & \textit{type}_{\textit{val}}\textbf{.store} \mid \textbf{call} \ \textit{idx}_{\textit{func}} \mid \\
 & \mid & \textbf{call\_indirect} \ \textit{type}_{\textit{func}} \mid \textbf{return} \mid \cdots \\
\textit{import} & ::= & (\texttt{"module"}, \texttt{"name"}) \\
\textit{export} & ::= & (\texttt{"name"}) \\
\textit{type}_{\textit{func}} & ::= & \textit{type}_{\textit{val}}{}^* \rightarrow \textit{type}_{\textit{val}}{}^* \\
\textit{type}_{\textit{val}} & ::= & \textbf{i32} \mid \textbf{i64} \mid \textbf{f32} \mid \textbf{f64} \\
\textit{idx}_{\textit{func}} & \in & \mathbb{N}
\end{array}
$$

Fig. 2: Abstract syntax of a simplified form of Wasm [18].

In summary, this paper contributes the following:

- The novel idea of leveraging targeted record and replay for execution-aware program reduction.
- A concrete implementation of this idea in RR-Reduce and Hybrid-Reduce, which are two program reduction techniques for Wasm.
- Empirical evidence that the use of execution behavior leads to more effective and efficient outcomes compared to existing program reduction techniques.
- We release our tools and data as open source, for others to reproduce and build on our results: https://github.com/sola-st/rr-reduce.

II. Background

A. WebAssembly

Wasm [1] is a compact binary format designed for efficient sandboxed execution in modern web browsers. Figure 2 shows a simplified abstract syntax of Wasm. A *module*, representing a single binary file, contains functions, global variables, tables, and memories. A *function* accepts parameters, declares local variables, executes instructions, and returns results. A *global* variable stores a single value accessible across all functions and can be mutable or immutable. A *table* maps indices to references of host objects or Wasm functions. A *memory* is a contiguous, byte-addressable, page-sized mutable array. These components can be imported from a host environment

using module and name pairs, or exported under one or more names for external access. Additionally, a module may include initialization data for tables and memories.

B. Wasm-R3

Wasm-R3 [12] is a record and replay technique for Wasm. It proceeds in three phases: record, reduce, and replay. The record phase captures the execution of an input Wasm program and produces a trace of events. The reduce phase minimizes this trace to a smaller sequence of events necessary for replay. The replay phase generates replay functions that reproduce the behavior of host functions using the reduced trace, and merges them with the input Wasm program to create a replay program. Of particular relevance to RR-Reduce is the replay phase. The replay program generated by Wasm-R3 has the following characteristics: (1) It preserves the individual functions in the input Wasm program as-is and only adds replay functions alongside them. This differs from other record and replay techniques [17] that modify the input program's functions, necessitated by the use of instrumentation during replay. (2) The replay functions are implemented as standard Wasm code. This approach makes the entire replay program standalone, allowing it to run on any Wasm engine and ensuring compatibility with various Wasm tools. These characteristics enable us to apply an existing program reduction technique (wasm-reduce) to the output of Wasm-R3, which would not be feasible with record and replay techniques lacking these properties.

III. Approach

This section presents RR-Reduce, an execution-aware program reduction technique for Wasm via record and replay. The approach leverages the ability of a record and replay technique to create replay functions that accurately reproduce an execution. The key idea is not to replay the entire program, but to selectively replay those parts of the program that are relevant to triggering the bug. We first give a high-level problem statement (Section III-A), provide an overview of our approach (Section III-B), introduce the running example (Section III-C), and finally present the details of each step (Sections III-D to III-H).

A. Problem Statement

We start by defining the problem we are addressing and by comparing it to problems addressed by previous work. The input to any program reduction technique is a program:

Definition 1 (Program)**.** A program $p \in P$ is a sequence of functions $f_1, ..., f_m$, where each function f_i consists of a sequence of instructions $i_1, ..., i_n$. The size of a program is the sum of the bytes of each instruction, i.e., $size(p) = \sum_{f_i \in p} \sum_{i_j \in f_i} bytes(i_j)$.

In practice, programs may contain other information (Figure 2), such as initial values of global variables, which we ignore for the purpose of concisely defining the problem. Our approach handles all elements of programs in Wasm, including functions, global variables, tables, and memories. We also assume that the program is self-contained and does not require any additional inputs for execution.

The motivation for reducing a program is that it has some property of interest, such as triggering a bug in a runtime engine. We assume to have an oracle that can check whether a program has this property. For example, such an oracle can be implemented by running the program and checking whether it triggers a specific bug in the runtime engine.

Definition 2 (Oracle)**.** An oracle $o : P \rightarrow \{true, false\}$ is a function that yields *true* if p has the property of interest, and *false* otherwise.

Previous program reduction techniques, such as [6], [7], [8], [9], [19], aim to reduce the size of a program while preserving the property of interest. This problem can be formulated as follows:

Definition 3 (Program reduction)**.** Given a program $p \in P$ and an oracle o, where $o(p) = true$, find a reduced program p' so that $o(p') = true$ and $size(p) > size(p')$.

In this work, we consider not only the program itself but also an execution of the program. Such an execution is available in many usage scenarios of program reduction, and it may provide valuable information for reducing a program. For example, when debugging a bug in a runtime engine that is triggered by a specific input program, considering the execution of the input program that leads to the bug may be helpful for finding a smaller input program. Our work exploits this insight by addressing a new, execution-aware variant of the program reduction problem:

Definition 4 (Execution-aware program reduction)**.** Given a program $p \in P$, an execution e of p that triggers the property of interest (typically a bug), and an oracle o, where $o(p) = true$, find a reduced program p' so that $o(p') = true$ and $size(p) > size(p')$.

To the best of our knowledge, we are the first to formulate and address the execution-aware program reduction problem in this way. The key difference between the execution-aware program reduction problem and the standard program reduction problem is that the former considers an execution of the input program, while the latter does not.

In addition to the problem statement above, a practical program reduction technique should also fulfill two additional requirements:

1) The reduced program should be small. Prior work on general test input reduction often strives for 1-minimality [6], i.e., no single constituent of the input can be removed without losing the property of interest. Concretely, this could, e.g., mean that a reduction technique removes as many functions from a program as possible, while still preserving the property. Our evaluation will show that considering the execution behavior during program reduction can lead to smaller reduced programs than existing techniques (Section IV-B).

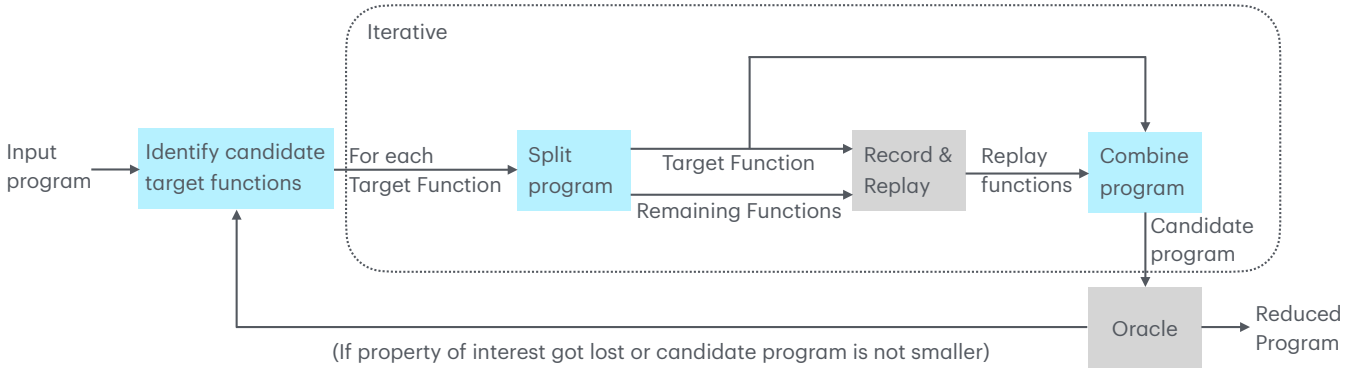

Fig. 3: Overview of RR-Reduce. Components in blue are introduced in this paper, while components in gray are external.

2) The reduction process should be reasonably fast. Reasoning about an execution of a program can be computationally expensive, yet offers the potential to reduce programs faster due to the additional information available. Our evaluation will show that an execution-aware reduction technique is not only more effective, but also more efficient than prior work (Section IV-C).

B. Overview

Figure 3 shows an overview of RR-Reduce. Given an executable program as input, RR-Reduce first heuristically identifies functions in the program that may be critical for triggering the bug, which the approach considers as candidate target functions (Section III-D). Then, RR-Reduce iterates over the candidate target functions one by one, selecting each as the target function. To start an iteration, the approach splits the input program into two parts: one containing the *target function* and another containing all *remaining functions* (Section III-E). The resulting partitioned program is then passed to an existing record and replay technique, which records the execution and produces *replay functions* (Section III-F). Importantly, we configure the record and replay technique to focus the replay on replaying only the target function. That is, the remaining functions are either removed or replaced with replay functions that mimic those parts of the behavior of the original functions that are necessary to accurately replay the target function. Given the replay functions, RR-Reduce combines the target function with the replay functions to produce a candidate for the reduced program (Section III-G). Finally, the oracle, also given as input, checks whether the candidate program still triggers the bug (Section III-H). If so, and if the candidate program is smaller than the input program, RR-Reduce returns the candidate program as the reduced program. Otherwise, the approach repeats the iteration with a different target function.

C. Running Example

Figure 4 shows a running example of RR-Reduce for a single iteration. On the left, we have an input program that consists of three functions: `a`, `b`, and `c`. Suppose an execution of this input program triggers a wrong-code bug in a Wasm engine, which results in a stack trace that contains the `c` function. Because `c` is in the stack trace, RR-Reduce heuristically identifies `c` as the target function. Next, RR-Reduce splits the input program into two parts: one containing the target function `c` and another containing the remaining functions `a` and `b`. Then, RR-Reduce uses an existing record and replay technique, Wasm-R3 (Section II-B), to produce replay functions that accurately replay the execution of the target function `c`. This is achieved in three phases: record, reduce, and replay. The resulting replay functions mimic the behavior of the remaining functions `a` and `b` in the input program. Our approach combines these replay functions with the target function `c` to produce a candidate program, and checks using the oracle whether the candidate program still triggers the bug in the Wasm engine and is smaller than the input program.

D. Identifying Candidate Target Functions

The first step of the approach is to identify a target function that triggers the bug. Various heuristics could be used for this purpose. For compiler-crash bugs in Wasm engines, our approach identifies the target function using error messages emitted by the compiler that specify which functions were being compiled at the time of the crash. For wrong-code bugs, which typically manifest as runtime traps instead of the expected program execution, RR-Reduce identifies relevant functions through the program's stack trace captured when traps are triggered. This heuristic stems from the observation that functions appearing in stack traces are often critical for both reproducing and diagnosing wrong-code bugs [20], [21], [22].

To realize this heuristic, RR-Reduce maintains three sets of functions, where each function is identified by its function index: The *All* set, with all functions in the input program. The *Dynamic* set, with all functions that are executed in the given execution, which we obtain via a simple dynamic analysis. The *Heuristic* set, with functions deemed most likely to be relevant for reproducing an engine bug. To compute the heuristic set, the approach performs a string search over the output of the engine compilation and execution, searching for

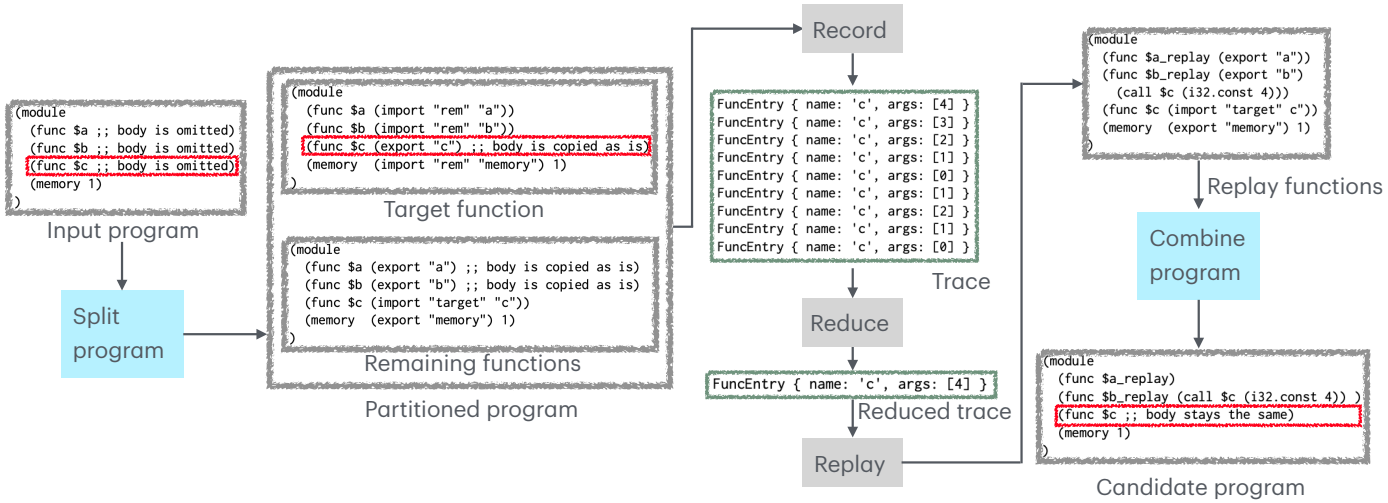

Fig. 4: Running example of RR-Reduce for a single iteration. We assume the target function is already selected by the previous step. Red rectangles denote the target function.

function indices. Given the three sets of functions, RR-Reduce enumerates candidates for the target function by prioritizing the *Heuristic* set, then the *Dynamic* set, and finally the *All* set.

E. Splitting the Program Into Target Function and Remaining Functions

Next, RR-Reduce takes an input program and the index of the candidate target function as an input, and outputs a partitioned program, which consists of two newly created Wasm binaries: one containing the target function and the other containing the remaining functions. To do so, the approach iterates over the program elements in the Wasm binary of the input program, performing different actions based on the type of each program element. For functions, RR-Reduce copies each function into one of the two new binaries: For the target function, it is copied to the binary that contains the target function. For the remaining functions, they are copied to the binary that contains the remaining functions. In addition to this copying, RR-Reduce performs additional bookkeeping to adjust the exports and imports of the newly created binaries. Previously, all the functions were inside a single program, so no exports and imports were needed for them to call each other. In contrast, in the partitioned program, each function is exported by one binary and then imported by the respective other binary. For globals, tables, and memories, RR-Reduce always copies them to the binary containing the remaining functions, and then shares them between the two binaries via exporting and importing.

To see how this looks in practice, refer to our running example in Figure 4, which shows how the input program gets split into two parts. The input program on the left consists of three functions: `a`, `b`, and `c`. On its right, the partitioned program consists of two binaries: The binary shown at the top, which contains the target function, and the binary shown at the bottom, which contains all the remaining functions of the input program. Suppose we run RR-Reduce on this program with the target function set as `c`. This means that function `a` and function `b` belong to the remaining functions. We now iterate over program elements in the input program, starting from the functions. For the `a` and `b` functions, as they belong to the remaining functions, we copy them to the binary at the bottom. We also export `a` with name "a" and `b` with name "b". In the binary at the top, we add imports with module "rem", name "a" and "b" respectively. Next, for the `c` function, as it is the target function, we copy it to the binary at the top. We also export `c` with name "c". In the binary at the bottom, we add an import with module "target", name "c". Finally, we move on to the memory section. As the two binaries should share the same memory, we copy the memory to the binary at the bottom. We also export memory with name "memory". In the binary at the top, we add an import with module "rem", name "memory".

F. Record and Replay for Program Reduction

Next, RR-Reduce applies the existing record and replay technique Wasm-R3 to the partitioned program. This step generates replay functions that mimic the relevant behavior of the remaining functions. A key difference between RR-Reduce and standard record and replay techniques is that RR-Reduce generates replay functions that replace functions belonging to the input program, instead of only reproducing side effects of the host environment. This unlocks the use of record and replay for program reduction, as removing and replacing functions of the input program potentially reduces the program size.

Although Wasm-R3 (Section II-B) itself is not a contribution of this paper, to better convey how RR-Reduce works, we illustrate how Wasm-R3 works for our running example in Figure 4. As a result of the split, we have a partitioned program with its two constituent parts. Assume that `c` is the target function, and `a` and `b` are the remaining functions; `a` calls `b`, and `b` calls `c`. Suppose additionally that the `c`

function contains recursive calls to itself. The record-reduce-replay part of the figure shows the three phases of Wasm-R3 applied to the partitioned program. When Wasm-R3 is run with this partitioned program, it first records the target function's execution into a trace. In the execution, the `c` function is called with an argument of 4 from the `b` function. This leads to recursive calls of the `c` function, resulting in 9 function entry events in the trace. However, as not all of these events are needed to replay the original execution of the target function, Wasm-R3 reduces the events to one, resulting in just a single function entry event with an argument of 4 in the reduce phase. Lastly, in the replay phase, Wasm-R3 transforms this reduced trace into a replay function, which calls the `c` function with the argument 4.

G. Combining Target Function and Replay Functions

After generating the replay functions, RR-Reduce combines the replay functions and the target function to produce a candidate program. This is possible as the replay functions are identical to the remaining functions they replace in terms of imports and exports. RR-Reduce then statically links and combines the replay functions and the target function into a single candidate program. This process typically results in a reduced program as: (1) Most of the remaining functions do not directly interact with the target function and are therefore eliminated. (2) For the remaining functions that do interact with the target function, their replay counterparts are generally smaller, as they only need to replicate the behavior relevant to the target function, not their entire behavior.

To illustrate this step, consider our running example in Figure 4 again. Replay functions do not contain their original bodies anymore; they contain the replay of the interactions with the target function. Despite these changes, they are still exported with the same names, allowing RR-Reduce to statically link and combine them. The resulting candidate program is presented on the right. Comparing the candidate program with the input program, we see why RR-Reduce can act as an effective program reduction technique. The target function is preserved, while one of the remaining functions has an empty body now, and the other remaining function has only two instructions.

H. Validation of the Candidate Program

Finally, RR-Reduce validates the candidate program using the oracle. Oracles are user-supplied scripts that check whether a program triggers the bug. Although the oracle can be any script, in our evaluation, we use a simple Python script that runs the candidate program once in an engine with a bug and once in an engine without the bug. Then, the oracle checks the return code, stdout, and stderr of the two runs to determine whether the candidate program still triggers the bug in the engine with the bug and terminates normally in the other engine. If the candidate program still triggers the bug, and if the candidate program is smaller than the input program, RR-Reduce returns the candidate program as the reduced program. Otherwise, the approach repeats the process with a different target function. If no reduced program is found after trying all possible target functions, RR-Reduce returns the input program as the result.

I. Combination with Existing Techniques into Hybrid-Reduce

The approach, as described so far, can be used as a stand-alone program reduction technique, which we call RR-Reduce. In addition, RR-Reduce can be combined with existing program reduction techniques to potentially produce even smaller reduced programs. We realize such a combination in Hybrid-Reduce, which feeds the output of RR-Reduce into the existing wasm-reduce [10]. Choosing between RR-Reduce and Hybrid-Reduce is a trade-off between obtaining a reduced program that keeps the target function as-is and a potentially even smaller reduced program. We study this trade-off in detail in our evaluation, which compares both approaches against the state of the art.

IV. Evaluation

We evaluate RR-Reduce by addressing the following three research questions:

- **RQ1: Effectiveness**. How effective are RR-Reduce and Hybrid-Reduce in reducing input programs?
- **RQ2: Efficiency**. How efficient are RR-Reduce and Hybrid-Reduce in terms of the time they take?
- **RQ3: Qualitative Analysis**. How do existing approaches and RR-Reduce qualitatively differ in their approach toward program reduction?

A. Experimental Setup

a) Dataset: As there is no previously available benchmark to evaluate Wasm program reduction techniques, we collect 28 Wasm programs that reveal 13 unique bugs in three Wasm engines: Wizard [23], WasmEdge [24], and WAMR [25]. To construct the benchmark, we first gather 21 out of 27 Wasm programs from the Wasm-R3-Bench dataset [12], which are able to trigger nine bugs in the Wizard engine, and all 25 Wasm programs from the WASMaker paper [3], which trigger four bugs in WAMR and WasmEdge. Out of the 21+25=46 candidate programs, we exclude 18 programs: 17 that use the SIMD extension of Wasm, which is currently not supported by Wasm-R3, and one that contains only a single function, i.e., there is nothing reduce for our approach. Table I lists the resulting 28 Wasm programs. In addition to the programs themselves, we provide for each program an oracle script that checks whether a reduced program still triggers the same bug.

b) Metrics: We measure the size of a Wasm program in terms of its code size, which is the size of the code section of the Wasm program. This excludes the size of the data section, which contains initialization data for the memory, and the size of the custom section, which contains debug symbols. Focusing on the code size is motivated by the fact that the code section is what engine developers typically focus on when debugging engine bugs. For RR-Reduce, we report two kinds of code sizes: the "All" code size, which includes all functions

TABLE I: Benchmark used to evaluate RR-Reduce. "Engine crash" refers to cases where the engine raises an error before executing the Wasm module. "Wrong code" refers to cases where the engine loads and compiles the Wasm module without errors but then deviates from the expected execution.

Name	Faulty engine	Fixed by	Kind	Code size
wasmedge#3018	WasmEdge	93fd4ae	Wrong code	1,913
wamr#2789	WAMR	718f067	Engine crash	17,604
wasmedge#3019	WasmEdge	93fd4ae	Wrong code	19,098
wamr#2862	WAMR	0ee5ffc	Wrong code	19,727
wamr#2450	WAMR	e360b7a	Engine crash	24,482
wasmedge#3076	WasmEdge	93fd4ae	Wrong code	31,365
mandelbrot	Wizard	0b43b85	Wrong code	64,515
pathfinding	Wizard	ccf0c56	Wrong code	180,026
pacalc	Wizard	81555ab	Wrong code	238,902
wasmedge#3057	WasmEdge	93fd4ae	Wrong code	243,564
guiicons	Wizard	6d2b057	Wrong code	285,840
rtexviewer	Wizard	708ea77	Engine crash	296,617
rfxgen	Wizard	6d2b057	Wrong code	378,918
riconpacker	Wizard	6d2b057	Wrong code	398,627
rguistyler	Wizard	6d2b057	Wrong code	410,845
rguilayout	Wizard	6d2b057	Wrong code	416,692
jqkungfu	Wizard	4e3e221	Engine crash	487,607
bullet	Wizard	f7aca00	Engine crash	536,115
funky-kart	Wizard	6d2b057	Wrong code	607,293
sqlgui	Wizard	6d2b057	Wrong code	628,046
hydro	Wizard	708ea77	Engine crash	719,538
figma-startpage	Wizard	33ec201	Engine crash	882,961
sandspiel	Wizard	ccf0c56	Wrong code	919,085
parquet	Wizard	33ec201	Engine crash	1,731,592
commanderkeen	Wizard	bc135ad	Wrong code	3,914,616
jsc	Wizard	6d2b057	Wrong code	4,342,199
boa	Wizard	6d2b057	Wrong code	5,198,069
ffmpeg	Wizard	4e3e221	Engine crash	5,356,751

in the binary, and the "Target" code size, which is the size of the target function. The "Target" is more relevant for engine developers, as they only need to inspect the target function which is responsible for triggering the bug.

c) Baselines: We compare RR-Reduce against two baselines: wasm-reduce [10] version 117, and wasm-shrink [11] version 1.227.0. To our knowledge, wasm-reduce and wasm-shrink are the only existing program reduction techniques for Wasm, and hence, the current state of the art. Both baselines are designed specifically for Wasm, following the style of C-Reduce [19]. Wasm-reduce is part of the Binaryen toolchain [26]. It interleaves semantics-destroying reductions, such as replacing a node with its child, and semantics-preserving optimizations. Wasm-shrink is part of the wasm-tools toolchain [27]. It also interleaves destructive reductions, such as deleting entire function bodies, with optimization reductions. Like our approach, wasm-reduce and wasm-shrink take as an input a program to reduce and an oracle script. Unlike RR-Reduce, neither of the two baselines are execution-aware, i.e., they do not make use of execution behavior of input programs.

d) Implementation and Hardware: We implement RR-Reduce through a combination of Python, Rust, and JavaScript, building on several libraries and tools in the Wasm ecosystem. The first step is to identify candidates for the target function (Section III-D). We use the existing tool, wasm-tools objdump [27], to compute the *All* set, a dynamic analysis based on the Wizard engine [13] to compute the *Dynamic* set, and implement the computation of the *Heuristic* set in Python. Next, our implementation splits the input program into two Wasm binaries (Section III-E), which we implement in Rust. In addition to two newly created Wasm binaries, our implementation also creates JavaScript glue code to dynamically link the two Wasm binaries. To resolve the circular dependency between the two binaries, the JavaScript code creates a closure that calls the target function, and instantiates the replay binary with this closure as an import. The record and replay step uses the existing Wasm-R3 [12] (commit hash 79b310b). Finally, our implementation combines the target function with the replay functions (Section III-G) using the wasm-merge tool [28]. To speed up the reduction process, our implementation tries to reduce the input program for different target functions in parallel. We ran all experiments on an Ubuntu 24.04 LTS system with an Intel Core i9-13900K (32 logical cores) and 192 GB of DRAM.

B. RQ1. Effectiveness

We evaluate the effectiveness of our approach by applying RR-Reduce and Hybrid-Reduce, as well as the two baselines, to the Wasm programs in Table I. We run each technique either until it terminates or a timeout of 24 hours is reached. We run eight reduction tasks in parallel, allowing each task to use a maximum of four logical cores. This ensures a fair comparison between wasm-reduce, RR-Reduce, and Hybrid-Reduce, which all exploit parallelism. [1]

The left-hand side of Table II shows the results. On average, RR-Reduce reduces programs to 6.67% of their original code size (using the "All" size) and to 1.20% when considering the "Target" size. For comparison, the baseline reduction tools reduce programs to 12.88% of their original size (wasm-shrink) and 0.43% (wasm-reduce). That is, RR-Reduce alone clearly outperforms one of the two baselines, and is competitive with the other.

The real strength of RR-Reduce is revealed when it is combined with existing program reduction techniques into a hybrid approach. Hybrid-Reduce reduces the input program to 0.13% of its original size, which is a 3.42× improvement over the current state of the art, wasm-reduce. Hybrid-Reduce achieves this by giving the best result in 23 cases, of which six are ties with the current state of the art. The benefit of the Hybrid-Reduce is not only in its better average performance, but also in its effectiveness on programs where the current state of the art struggles. Table III highlights the difference. Specifically, wasm-shrink produces programs larger than 100KB in 15 cases, and wasm-reduce does so in three cases. In contrast, all Hybrid-Reduce-reduced programs are well below 100KB, and often even further.

[1] Wasm-shrink does not exploit parallelism.

TABLE II: Comparison of wasm-shrink, wasm-reduce, RR-Reduce, and Hybrid-Reduce. The most reduced programs are marked in bold. "Average" means geometric mean for effectiveness and arithmetic mean for efficiency. For RR-Reduce, "All" measures all functions in the reduced program, while "Target" measures only the target function in the reduced program.

Name	Effectiveness: Reduced code size (lower is better)					Efficiency: Time taken (s) (lower is better)			
	Baselines		Our Work			Baselines		Our Work	
	Wasm-shrink	Wasm-Reduce	RR-Reduce		Hybrid-Reduce	Wasm-shrink	Wasm-Reduce	RR-Reduce	Hybrid-Reduce
			All	Target					
wasmedge#3018	4.65%	1.25%	23.73%	17.09%	**0.58%**	128	23	25	51
wamr#2789	**0.05%**	**0.05%**	2.33%	0.59%	**0.05%**	24	6	1,200	1,210
wasmedge#3019	7.71%	**0.06%**	3.75%	0.92%	**0.06%**	10	4,194	30	56
wamr#2862	2.95%	**0.18%**	9.27%	7.12%	**0.18%**	299	54	136	187
wamr#2450	11.48%	**0.03%**	3.33%	1.73%	**0.03%**	34	31	45	58
wasmedge#3076	27.33%	0.04%	2.81%	0.33%	**0.04%**	86,400	1,017	783	799
mandelbrot	27.00%	21.43%	94.70%	2.57%	**0.43%**	15,178	86,400	235	73,482
pathfinding	10.57%	0.04%	31.18%	0.12%	**0.03%**	3,721	3,627	964	1,155
pacalc	17.84%	0.22%	14.57%	0.89%	**0.08%**	615	1,014	15	21,424
wasmedge#3057	26.48%	<0.01%	2.52%	0.17%	**<0.01%**	86,400	2,598	2,034	1,964
guiicons	42.38%	**11.65%**	60.40%	42.01%	11.84%	1,284	29,266	75	20,458
rtexviewer	1.36%	0.17%	3.44%	2.28%	**0.02%**	774	196	485	640
rfxgen	30.59%	8.96%	54.17%	30.21%	**8.72%**	24,208	9,475	108	13,283
riconpacker	35.89%	**8.23%**	42.65%	35.76%	8.39%	1,506	68,186	10	20,522
rguistyler	36.34%	8.16%	68.11%	35.20%	**8.15%**	2,126	34,934	65	9,681
rguilayout	38.36%	**8.10%**	62.92%	37.92%	10.52%	37,380	34,244	82	10,538
jqkungfu	3.13%	2.78%	5.25%	4.56%	**0.26%**	86,400	1,241	9	183
bullet	62.64%	6.76%	2.75%	0.06%	**<0.01%**	86,400	86,400	13,585	17,276
funky-kart	36.11%	5.76%	19.65%	17.59%	**5.44%**	86,400	86,400	329	86,400
sqlgui	21.66%	12.23%	13.99%	**5.39%**	11.16%	12,192	13,534	24	50,425
hydro	0.88%	0.48%	2.59%	1.44%	**0.13%**	86,400	541	2,237	2,042
figma-startpage	17.86%	**<0.01%**	3.24%	0.02%	**<0.01%**	86,400	123	12	25
sandspiel	96.00%	**<0.01%**	0.13%	0.03%	<0.01%	2,570	84,626	897	952
parquet	25.44%	**<0.01%**	0.94%	<0.01%	**<0.01%**	86,400	216	18	25
commanderkeen	39.51%	33.44%	1.35%	0.02%	**<0.01%**	86,400	86,400	788	5,945
jsc	64.51%	6.36%	6.34%	3.94%	**0.79%**	86,400	86,400	65	15,633
boa	8.90%	7.21%	1.36%	0.94%	**0.64%**	28,437	86,400	110	3,140
ffmpeg	4.47%	<0.01%	0.62%	0.11%	**<0.01%**	86,400	982	48	150
Average	12.88%	0.43%	6.67%	1.20%	**0.13%**	38,603	28,876	871	12,775

TABLE III: Programs for which the current state of the art struggles. Reduced programs larger than 100 KB are shown in red, and those below 1 KB in blue.

	Input	Wasm-reduce	Hybrid-Reduce
mandelbrot	64 KB	13 KB	276 B
bullet	536 KB	36 KB	53 B
hydro	720 KB	3.4 KB	939 B
commanderkeen	3.9 MB	1.3 MB	158 B
jsc	4.3 MB	276 KB	34 KB
boa	5.2 MB	374 KB	33 KB

RR-Reduce reduces programs to 1.20% of their original size. Hybrid-Reduce further reduces programs down to 0.13%, which means it outperforms the state of the art by 3.42× in terms of effectiveness.

C. RQ2. Efficiency

Together with effectiveness, efficiency is another important goal of program reduction techniques. This is because, if program reduction techniques take unreasonable time to reduce the program, they are not helpful to engine developers. The right-hand side of Table II shows the efficiency results, which indicate the time taken to reduce the bug-inducing programs. RR-Reduce takes 14.5 minutes to reduce the bug-inducing programs, using the arithmetic mean. Hybrid-Reduce takes about 3.5 hours, while the current state of the art, wasm-reduce, takes eight hours. Thus, Hybrid-Reduce achieves a 2.26× improvement in terms of time over the current state of the art, while also being more effective (see RQ1). In comparison to wasm-shrink, which takes ten hours, Hybrid-Reduce achieves a 3.02× improvement.

RR-Reduce and Hybrid-Reduce are able to achieve this speedup due to their heuristic identification of the target function. Among the 28 programs in the evaluation set, the target function is identified in the *Heuristic* set in 13 cases, the *Dynamic* set in 6 cases, and the *All* set in 9 cases. As

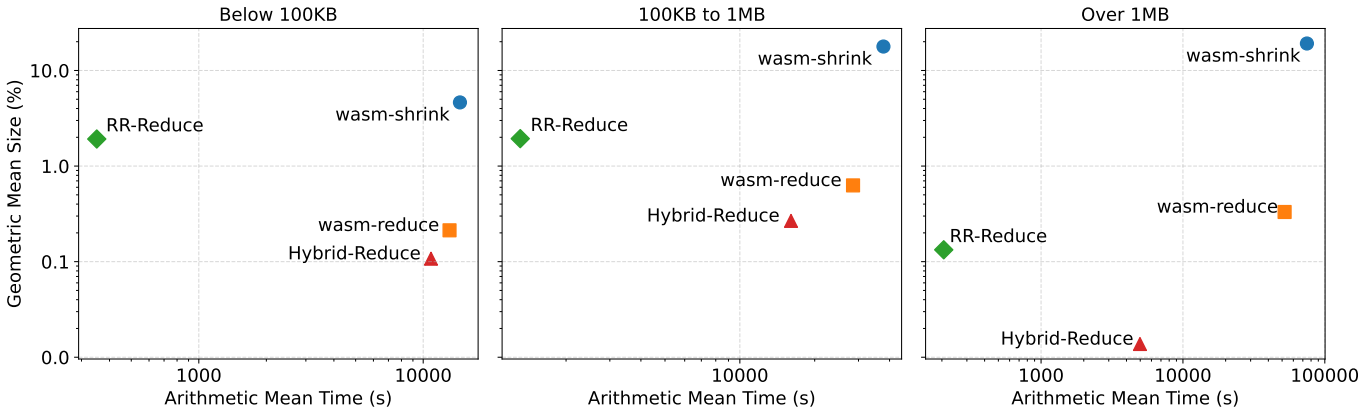

Fig. 5: Tradeoff between time taken to reduce programs and the size of the reduced programs.

the average size of the *Heuristic* set is 2.68, the *Dynamic* set is 277.04, and the *All* set is 1,995.57, identifying the target function in the *Heuristic* set greatly reduces the search space. Even if the heuristic fails, RR-Reduce can fall back to the *Dynamic* or *All* sets, losing some efficiency but without affecting its effectiveness.

To clearly show the tradeoff between time and reduction rate, Figure 5 plots time on the x-axis and size of the reduced code ("Target" size in the case of RR-Reduce) on the y-axis. We divide the evaluation set into three groups based on input code size: the below 100KB group, which contains seven programs; the 100KB to 1MB group, which contains 16 programs; and the over 1MB group, which contains five programs. In all three groups, Hybrid-Reduce outperforms wasm-reduce and wasm-shrink in both time and reduction rate. RR-Reduce is the fastest in all three groups, while outperforming wasm-shrink in terms of effectiveness in all three groups and outperforming wasm-reduce in terms of effectiveness in the over 1MB group. We envision both of our approaches being useful depending on the usage scenario: For fast isolation of the bug-triggering function within minutes, RR-Reduce is the way to go; if a user is willing to wait a few hours, then Hybrid-Reduce is preferable.

> RR-Reduce and Hybrid-Reduce reduce programs in 14.5 minutes and 3.5 hours, respectively, which corresponds to an 33.15× and 2.26× improvement over the state of the art in terms of efficiency.

D. RQ3. Qualitative Analysis

Our final research question qualitatively analyzes the differences between our approaches and the existing program reduction techniques. Large programs are where the engine developers benefit the most from the help of program reduction techniques, because for them, developers need to spend the most time. We hence focus on the four programs that are over 1MB in size. They reveal four different bugs, providing a demonstration of the different approaches employed by program reduction techniques. We exclude the analysis of jsc, as it triggers the same bug as boa, and much of the analysis would be redundant.

Boa, which is 5.2MB in size, triggers a wrong-code bug[2] that gets triggered when the interpreter tries to interpret a jump with an offset bigger than 32,768 bytes. Wasm-reduce reduces the input program to 317KB, which is still too large to manually debug. It also contains 87 functions besides the function where the jump happens, which further complicates debugging. Wasm-reduce cannot simply remove these functions because other functions are necessary for the jump to happen. In contrast, RR-Reduce takes a different approach: It deletes the bodies of 453 executed functions and replaces the bodies of eight functions with replay functions. Among the nine functions that are executed in the reduced program, seven are short functions that update the state, and one is the entry to the reduced program, which calls the bug-triggering functions. The remaining one is the bug-triggering function, which is identical to the function in the input program. The biggest difference between the existing approaches and RR-Reduce is most evident in its treatment of the functions that interact with the target function to create the necessary state to trigger the bug. In the existing approaches, every code that has been executed to create such a state is preserved. However, RR-Reduce just keeps the effects of the other code to make the jump happen, which yields a program with 48,862 bytes in terms of the target size. Hybrid-Reduce reduces the program even further to 33,358 bytes.

Commanderkeen, which is 3.9MB in size, triggers a subtle wrong-code bug[3] in the JIT compiler. Wasm-reduce struggles the most, only reducing the input program to 1.3MB, which is hard for a human to manually debug. The bug is a subtle register allocation problem that occurs within the engine's implementation of the `call_indirect` instruction. RR-Reduce selects the function where the behavior diverges as a target function and replaces the bodies of the functions that interact with the target function with replay functions, while deleting

[2]Fixed by https://github.com/titzer/wizard-engine/commit/6d2b057
[3]Fixed by https://github.com/titzer/wizard-engine/commit/bc135ad

all the other functions, resulting in a reduced program of 782 bytes in terms of the target size. Hybrid-Reduce reduces the further down to 158 bytes, shown in Figure 1.

Ffmpeg, which is 5.3MB in size, triggers a compiler-crash bug[4]. As this is a compiler-crash bug, most of the functions can be safely deleted except the bug-triggering function. Thus, wasm-reduce is effective in this case, reducing the input program to 479 bytes in 982 seconds. Hybrid-Reduce achieves an even smaller result of 45 bytes in only 150 seconds.

Parquet, which is 1.7MB in size, triggers a trivial compiler-crash bug[5] in the Wizard engine. For this bug to be triggered, it is sufficient to contain a single memory declaration with 64-bit addresses. All other code can be removed as long as the Wasm module retains the offending memory declaration. Thus, wasm-reduce is effective, producing a reduced program that contains a single function with a single nop instruction in 216 seconds. Hybrid-Reduce achieves the same reduction but quicker; it takes only 25 seconds to obtain the reduced output.

> For two programs over 1MB causing wrong-code bugs, only Hybrid-Reduce is effective enough to facilitate the manual debugging while still being efficient. For two programs over 1MB causing compiler-crash bugs, the current state-of-the art, wasm-reduce, is effective, but Hybrid-Reduce gives even better results faster.

V. Discussion

A. Threats To Validity

a) Internal validity: Our effectiveness results are influenced by the timeout setting we used (24 hours). Thus, effectiveness would improve if we allow the tools to run longer. However, we believe 24 hours is a reasonable assumption on what developers will tolerate when debugging a single bug-triggering program. This timeout is also used in other recent program reduction work [29].

b) External validity: Our results are limited to the 28 programs in our evaluation set and may not generalize to all Wasm applications. However, our evaluation set covers diverse real-world use cases, including programming-language runtimes, media applications, video games [12], and automatically generated Wasm programs [3].

B. Limitations

RR-Reduce uses simple string search over the engine's output to heuristically identify a target function that triggers the bug. This heuristic can be effective for engine crash bugs and wrong-code bugs that lead to a runtime error. It is not effective for wrong-code bugs that do not manifest as observable changes in behavior. In such cases, RR-Reduce can fall back to the Dynamic set and All set at the cost of longer reduction time, yet still produce the same final minimized output. Another limitation is that RR-Reduce currently supports reduction of Wasm programs up to the Wasm 2.0 specification, excluding SIMD, due to the limitations of Wasm-R3. It also relies on the correctness and performance of Wasm-R3. Addressing more Wasm extensions and improving performance will require further engineering of the Wasm-R3 toolchain.

[4]Fixed by https://github.com/titzer/wizard-engine/commit/4e3e221

[5]Fixed by https://github.com/titzer/wizard-engine/commit/33ec201

C. Generalization to Other Programming Languages

Conceptually, RR-Reduce can be applied to other programming languages. There are a few conditions for this to be possible: (1) A record and replay implementation must exist. (2) The record and replay system must support selective record and replay. (3) The output of selective record and replay must be a regular program.

VI. Related Work

a) Dynamic Analysis for WebAssembly: There are several dynamic analysis techniques for Wasm. Wasabi [18] designs and implements bytecode-level dynamic instrumentation for Wasm to enable diverse dynamic analyses. Wizard [30] supports non-intrusive instrumentation for Wasm by engine-level dynamic instrumentation. Wasm-R3 [12] is a record and replay technique for Wasm that enables the generation of executable, standalone Wasm benchmarks. Wemby [31] utilizes dynamic analysis to detect memory corruption bugs in Wasm. Our work contributes to the field by utilizing the execution behavior of Wasm programs to improve Wasm program reduction.

b) Record and Replay: Record and replay is a well-established research area that has been studied in multiple domains, including architectural support [32], OS-level implementations [33], user-space implementations [15], language-runtime integrations [34], JavaScript benchmark generation [16], and Wasm benchmark generation [12]. Selective record and replay techniques [35], [36], [17], [20], [37], which record and replay only part of an execution, are particularly relevant to us. All of these techniques utilize selective record and replay with different goals in mind. To our knowledge, however, RR-Reduce is the first to apply selective record and replay to program reduction.

c) Test Input Reduction: Delta debugging (DD) [6] pioneered automated test input reduction. Hierarchical Delta Debugging (HDD) [7] adapts DD for hierarchical inputs such as programming languages. Generalized Tree Reduction (GTR) [8] generalizes HDD to support arbitrary tree transformations and specializes these transformations by learning from a corpus of example data. Perses [9] uses formal syntax of the program to make language-agnostic program reduction more effective, efficient, and general. Vulcan [38] utilizes diverse program transformations to further reduce the output of language-agnostic program reducers. PPR [29] minimizes pairs of programs rather than a single program. LPR [39] leverages LLM for program reduction. None of these approaches, however, exploits the execution behavior of the input program to guide reduction. In contrast, our approach is execution-aware; it leverages an execution of the program to achieve more effective and efficient outcomes.

There is one approach that we are aware of that exploits execution behavior to improve delta debugging. Fast-Reduce, one of the program reduction techniques introduced in [19], uses run-time information obtained from instrumentation. One transformation that Fast-Reduce applies is to inline calls to functions with their dynamic effect, which allows removal of the called function after all its call sites are inlined. Our approach differs from Fast-Reduce in the following ways: (1) Fast-Reduce relies on dynamic analysis specifically built to integrate with the test case generator CSmith [40], whereas RR-Reduce employs a general-purpose record and replay tool; (2) Fast-Reduce selects an arbitrary final effect of the function to inline, whereas RR-Reduce records and replays every effect of the function across its multiple calls. (3) Fast-Reduce does not empirically outperform other techniques in terms of effectiveness, whereas the use of RR-Reduce achieves such outcomes. In summary, although Fast-Reduce and RR-Reduce share some similarities, we find RR-Reduce to be a more principled and effective realization of the idea of using execution behavior to improve program reduction.

VII. Conclusion

We present an execution-aware program reduction technique for Wasm that utilizes record and replay for the purpose of program reduction, with its concrete implementation in RR-Reduce and Hybrid-Reduce. A key insight is that record and replay of only part of the input program preserves the execution behavior of that part while either deleting or replacing the rest of the program. We evaluate RR-Reduce and Hybrid-Reduce on a set of Wasm programs and found it to be effective and efficient. We hope that RR-Reduce and Hybrid-Reduce pave the way for program reduction techniques to scale to much larger and more complex programs, extending the reach of occasions where Wasm engine developers can benefit from program reduction techniques. In addition, we hope that RR-Reduce and Hybrid-Reduce inspire the development of similar techniques in other languages by leveraging selective record and replay.

Data Availability

The artifact (source code, benchmark, evaluation outputs, and documentation) is available at https://github.com/sola-st/rr-reduce.

Acknowledgements

We thank Laurence Tratt and anonymous reviewers for their helpful feedback. This work was partially supported by NSF award #2148301, the WebAssembly Research Center, the National Research Foundation of Korea (NRF) (2022R1A2C2003660 and 2021R1A5A1021944), an Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (2024-00337703), Samsung Electronics Co., Ltd., by the European Research Council (ERC, grant agreements 851895 and 101155832), and by the German Research Foundation within the DeMoCo project.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 2017, pp. 185–200.

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

[3] S. Cao, N. He, X. She, Y. Zhang, M. Zhang, and H. Wang, "Wasmaker: Differential testing of webassembly runtimes via semantic-aware binary generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1262–1273. [Online]. Available: https://doi.org/10.1145/3650212.3680358

[4] 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*, 2024, pp. 1035–1047.

[5] N. Rao, E. Gilbert, T. Ramananandro, N. Swamy, C. L. Goues, and S. Fakhoury, "Diffspec: Differential testing with llms using natural language specifications and code artifacts," *arXiv preprint arXiv:2410.04249*, 2024.

[6] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183–200, 2002. [Online]. Available: https://doi.org/10.1109/32.988498

[7] G. Misherghi and Z. Su, "HDD: hierarchical delta debugging," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 142–151. [Online]. Available: https://doi.org/10.1145/1134285.1134307

[8] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 861–871, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:4652972

[9] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 361–371, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:3770204

[10] A. Zakai, "Fuzzing · webassembly/binaryen wiki," 2024, retrieved Oct 13, 2024. [Online]. Available: https://github.com/WebAssembly/binaryen/wiki/Fuzzing#reducing

[11] A. Crichton, "wasm-shrink - crates.io: Rust package registry," 2024, retrieved Oct 13, 2024. [Online]. Available: https://crates.io/crates/wasm-shrink

[12] D. Baek, J. Getz, Y. Sim, D. Lehmann, B. Titzer, S. Ryu, and M. Pradel, "Wasm-r3: Record-reduce-replay for realistic and standalone webassembly benchmarks," in *Proceedings of the ACM on Programming Languages: Object-Oriented Programming, Systems, Languages & Applications*, ser. OOPSLA '24, 2024.

[13] B. L. Titzer, "A fast in-place interpreter for webassembly," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563311

[14] H. Patil, C. L. Pereira, M. Stallcup, G. Lueck, and J. H. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:17445756

[15] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[16] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of javascript benchmarks," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 677–694. [Online]. Available: https://doi.org/10.1145/2048066.2048119

[17] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: a selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 488–498. [Online]. Available: https://doi.org/10.1145/2491411.2491447

[18] 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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1045–1058. [Online]. Available: https://doi.org/10.1145/3297858.3304068

[19] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:1025409

[20] M. Burger and A. Zeller, "Minimizing reproduction of software failures," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 221–231. [Online]. Available: https://doi.org/10.1145/2001420.2001447

[21] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 151–160. [Online]. Available: https://doi.org/10.1109/ICSME.2014.37

[22] Y. Gu, J. Xuan, H. Zhang, L. Zhang, Q. Fan, X. Xie, and T. Qian, "Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence," *Journal of Systems and Software*, vol. 148, pp. 88–104, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2018.11.004

[23] B. L. Titzer, "Wizard, An advanced Webassembly Engine for Research," https://github.com/titzer/wizard-engine, 2021, retrieved Februar 23, 2024. [Online]. Available: https://github.com/titzer/wizard-engine

[24] "WasmEdge," https://github.com/WasmEdge/WasmEdge, 2024, (Accessed 2024-10-11). [Online]. Available: https://github.com/WasmEdge/WasmEdge

[25] "WebAssembly Micro Runtime (WAMR)," https://github.com/bytecodealliance/wasm-micro-runtime, 2022, (Accessed 2022-04-11). [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[26] "Webassembly/binaryen: Optimizer and compiler/toolchain library for webassembly," https://github.com/WebAssembly/binaryen, 2024, retrieved April 3, 2024.

[27] "Cli and rust libraries for low-level manipulation of webassembly modules," https://github.com/bytecodealliance/wasm-tools, 2024, (Accessed 2024-10-04). [Online]. Available: https://github.com/bytecodealliance/wasm-tools

[28] A. Zakai, "Reintroduce wasm-merge," https://github.com/WebAssembly/binaryen/pull/5709, 2023, retrieved March 14, 2024.

[29] M. Zhang, Z. Xu, Y. Tian, Y. Jiang, and C. Sun, "PPR: pairwise program reduction," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 338–349. [Online]. Available: https://doi.org/10.1145/3611643.3616275

[30] B. L. Titzer, E. Gilbert, B. W. J. Teo, Y. Anand, K. Takayama, and H. Miller, "Flexible non-intrusive dynamic instrumentation for webassembly," *arXiv preprint arXiv:2403.07973*, 2024.

[31] O. Draissi, T. Cloosters, D. Klein, M. Rodler, M. Musch, M. Johns, and L. Davi, "Wemby's web: Hunting for memory corruption in webassembly," *Proc. ACM Softw. Eng.*, vol. 2, pp. 1326–1349, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:279473159

[32] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, p. 122–135, may 2003. [Online]. Available: https://doi.org/10.1145/871656.859633

[33] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen, "Revirt: Enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 211–224, 2002.

[34] O. Sahin, A. Aliyeva, H. Mathavan, A. Coskun, and M. Egele, "Randr: Record and replay for android applications via targeted runtime instrumentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 128–138.

[35] A. Orso and B. Kennedy, "Selective capture and replay of program executions," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1 – 7, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:11385424

[36] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, D. F. Redmiles, T. Ellman, and A. Zisman, Eds. ACM, 2005, pp. 114–123. [Online]. Available: https://doi.org/10.1145/1101908.1101927

[37] M. Hammoudi, B. Burg, G. Bae, and G. Rothermel, "On the use of delta debugging to reduce recordings and facilitate debugging of web applications," *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:9212076

[38] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun, "Pushing the limit of 1-minimality of language-agnostic program reduction," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 636–664, 2023. [Online]. Available: https://doi.org/10.1145/3586049

[39] M. Zhang, Y. Tian, Z. Xu, Y. Dong, S. H. Tan, and C. Sun, "LPR: large language models-aided program reduction," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, M. Christakis and M. Pradel, Eds. ACM, 2024, pp. 261–273. [Online]. Available: https://doi.org/10.1145/3650212.3652126

[40] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:868674