Semantic fuzzing of the Rust compiler and interpreters
Master's Thesis Project Description

Qian Wang
Supervised by Prof. Dr. Ralf Jung
Department of Computer Science, ETH Zurich

31 January 2023 to 31 July 2023

1 Introduction

The Rust programming language has seen increasing adoptions in recent years, notably in Firefox and Chrome web browsers, the Android operating system, and the Linux kernel. A primary appeal of Rust over the established systems programming languages C and C++ is that Rust programs without `unsafe` blocks are guaranteed to not contain any Undefined Behaviour (UB), such as buffer overflows and use-after-frees. However, `unsafe` is present somewhere in the transitive dependency trees of all but the most trivial Rust programs. `unsafe` is not meant to be avoided at all costs, but rather to be used carefully to build *safe abstractions* around them so that UB cannot be invoked for all possible well-typed Rust programs using the safe abstraction.

The observable behaviours of a UB-free program execution must be predictable (although not necessarily deterministic). To perform virtually any optimisations at all, the compiler must assume that UB never occur at program runtime, and preserve the observable program behaviours under all potential UB-free executions. This is already a difficult task, but the Rust compiler `rustc` faces two additional challenges: one is that Rust currently does not have a formal specification of its semantics, part of which details what is and what isn't UB. These are informally defined in documentations and RFCs, but they leave many edge cases under-specified. The second is that the Rust compiler is designed to produce a variety of Intermediate Representations (such as LLVM IR) as compilation outputs. The generated IR must preserve the original program behaviours, but the IRs may differ subtly in semantics, leading to miscompilations.

The Miri [1] and MiniRust [2] projects partly address the first problem. Miri is a Rust interpreter capable of executing Rust programs and detecting some classes of UB. Code authors can execute their code with Miri to see if a particular execution relies on any behaviours Miri considers UB. The `MiniRust` project aims to produce the formal operational semantics of a stripped-down version of Rust. The operational semantics can be transpiled and executed to observe program behaviours and detect UB.

However, there is currently no guarantee of equivalence between the runtime behaviour of a program compiled by `rustc`, the execution of `miri`, and the operational semantics defined by `MiniRust`, but we'd like to have some assurances that they are indeed the same. To address this problem, this project aims to build a fuzzer that generates random Rust programs - some of which may contain interesting edge cases - and then detect any difference in program behaviour when compiled or interpreted by different tools to expose their bugs.

2 Approach

The fuzzer consists of two main components. The first component is responsible for generating Rust programs in the form of Mid-level Intermediate Representation (MIR), and the second component performs differential testing using the Rust compiler with different backends, Miri, and potentially the MiniRust interpreter.

2.1 MIR generation

Rust's MIR originated as a stage in `rustc`'s compilation pipeline. It is a Control-flow graph (CFG) where each function body contains a list of *basic blocks*, in turn containing statements. Statements within a basic block are to be executed top-down with no branching. A basic block always ends with a *terminator* which can lead to multiple potential basic blocks. The program's control flow, such as if-statements, loops and function calls, are represented by the connections between basic blocks. In addition to basic blocks, MIR function bodies also contain declaration of local variables and their types.

The MIR encapsulates the semantics of a Rust program using a small set of primitives. For this reason, the MIR has been used to implement Miri, and the operational semantics produced by MiniRust is defined on a MIR-like language.

Due to its simplicity and relatively well agreed semantics, MIR is chosen as our fuzzer generation target.

MIR generation will be done in several steps. First, we generate a Control-flow graph skeleton, which contains functions and empty basic blocks. To simplify the subsequent steps, no back edges are generated. This is left as a part of the extensions.

Then we generate the user-defined types, such as `struct`s and `enum`s. These type definitions are not in the MIR, they will be supplied to `rustc` separately when we compile or run the MIR.

Next, we generate statements within the basic blocks, which is done top-down. Local variables are generated as needed. Since only UB-free programs can be meaningfully compiled by `rustc`, we need to guarantee UB-freedom of the generated program. There are two possible approaches: We could generate fully random MIR statements with little regard to semantics, and then use Miri as an oracle to filter out UB-containing programs. However, it is possible that the probability of UB-freedom of a random program is too small for this approach to be efficient. Alternatively, we could use static constraints during program generation to produce UB-free programs directly. The lack of back edges makes static analysis significantly simpler. The former approach will be evaluated first to determine if the latter, more complex approach is required.

There is a last and optional step to generate programs that potentially contain UB. If we used Miri as a filter in the previous step, we could simply use the ones that have been discarded. If we generated UB-free programs directly, then we need to mutate the MIR, especially at places where the static constraints were used to prevent UB. This should have a high chance of producing edge-case UB that may be handled incorrectly in one of the differential testing targets.

A UB-free program can still exhibit non-determinism, which may cause false positives in the later differential testing stage. Therefore our MIR generation must guarantee determinism in the observable program outputs. Transient non-determinism during execution which does not change the observed outputs is still allowed. One approach to test for non-determinism is to run them with Miri several times with different random seeds, and the ones that always produce the same outputs are likely to be semantically deterministic. Additionally, similar to UB filtering mentioned above, it may also be possible to statically prevent some patterns of non-determinism, or to transform a non-deterministic program into a deterministic one.

2.2 Differential testing

Differential testing is a technique where the same program is compiled and/or executed by different implementations of the language and observing the difference in program behaviours [3]. A difference in behaviour may indicate a bug in one of the implementations.

We are interested in differentially testing two types of behaviours: the execution of MIR programs free of UB, and the detection of UB of MIR programs potentially containing them.

The Rust compiler, `rustc`, is not a monolith. Instead, it supports different machine-code generation backends (commonly referred to as codegen backends). By far the most commonly used codegen backend is the LLVM backend (`rustc_codegen_llvm`), where `rustc` generates LLVM IR from the input Rust program, and LLVM then compiles the IR into executable machine code. But there are two additional codegen backends currently in `rustc`: `rustc_codegen_gcc` and `rustc_codegen_cranelift`, which generates executable machine code using GCC and Cranelift respectively.

Besides being compiled, MIR can be interpreted by Miri, which additionally checks for UB during execution.

For a UB-free test program, we expect identical program behaviours (bar any expected nondeterminism) across all 4 of the above execution methods, plus the fact that Miri do not report UB. If any of them deviates from the others, then it likely indicates a bug, which will then be manually investigated.

MIR can also be transpiled to MiniRust, which can then be transpiled into Rust code and executed. Like Miri, MiniRust defines the program states and actions considered UB and reports them. This MiniRust interpreter [4] is under active development and currently only supports a small fragment of the language. For this reason, MiniRust supoort is left as an extension goal.

Supporting MiniRust allows us to use it for differential testing using UB-free test programs. It also opens ups the possibility to compare Miri and MiniRust using potentially UB-containing programs, where we expect Miri and MiniRust to both report or not report Undefined Behaviours. A difference in UB reporting indicates a false positive or negative in either of these tools.

3 Core Goals

Designing UB-free MIR generation: Although some techniques to prevent UB in generated MIR have been mentioned above, they are certainly not fully sufficient. Cases may come up during the implementation that require special attention.

Implementation of the MIR generator: Generating UB-free and potentially UB-containing MIR programs as specified above.

Implementation of the differential testing framework: Given the generated MIR, the testing framework can plug this into `rustc` and `miri` to execute it.

Handling non-determinism: A UB-free program without any input may still exhibit non-determinism, such as reading the address of a pointer. The MIR generator should not generate programs with non-deterministic outputs.

4 Extension Goals

Generating back edges: This allows us to generate loops in MIR, and potentially recursion, which widens the test surface.

Generating Rust source code from MIR: This allows us to test alternative Rust compilers that do not use MIR, such as `gcc-rs` and `mrsutc`.

MiniRust support: Add the MiniRust interpreter as a differential testing target.

5 Schedule

This 20-week schedule is for guidance. The project start date is 31 January and the report submission deadline is 31 July. Ample time is left should any unexpected complication come up.

MIR generator	6 weeks
Differential tester	4 weeks
Evaluation	1 week
Extensions	5 weeks
Report writing	4 weeks

References

[1] "Miri," 2016. [Online]. Available: https://github.com/rust-lang/miri

[2] R. Jung, “Announcing: Minirust,” 2022. [Online]. Available: https://www.ralfj.de/blog/2022/08/08/minirust.html

[3] W. M. McKeeman, “Differential testing for software,” *Digit. Tech. J.*, vol. 10, pp. 100–107, 1998.

[4] R. Schneider, “minirust-tooling,” 2022. [Online]. Available: https://github.com/memoryleak47/minirust-tooling