RustSmith: Random Differential Compiler Testing for Rust

Mayank Sharma
Imperial College London
UK

Pingshi Yu
Imperial College London
UK

Alastair F. Donaldson
Imperial College London
UK

ABSTRACT

We present RustSmith, the first Rust randomised program generator for end-to-end testing of Rust compilers. RustSmith generates programs that conform to the advanced type system of Rust, respecting rules related to *borrowing* and *lifetimes*, and that are guaranteed to yield a well-defined result. This makes RustSmith suitable for differential testing between compilers or across optimisation levels. By applying RustSmith to a series of versions of the official Rust compiler, `rustc`, we show that it can detect insidious historical bugs that evaded detection for some time. We have also used RustSmith to find previously-unknown bugs in an alternative Rust compiler implementation, `mrustc`. In a controlled experiment, we assess statement and mutation coverage achieved by RustSmith vs. the `rustc` optimisation test suite.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Maintaining software*; **Software testing and debugging**.

KEYWORDS

Compiler testing, differential testing, fuzzing, mutation testing, Rust

ACM Reference Format:
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 (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604919

1 INTRODUCTION

The Rust programming language is growing rapidly in popularity. Through an innovative type system, Rust statically eliminates classes of memory-related bugs that are the bane of unsafe languages such as C and C++. Because of these safety guarantees, Rust is being used for the development of key infrastructure. It is thus critical that the Rust compiler, `rustc`, is thoroughly tested. To increase the thoroughness of `rustc` testing, we have built RustSmith, the first randomised generator of Rust programs suitable for differential compiler testing. The name, and some of the design of RustSmith is inspired by Csmith [20], a widely-used generator of C programs. RustSmith produces well-formed programs that conform to the advanced type system of the language. Generated programs can be used to cross-check `rustc` at different optimisation levels, or against other Rust compilers such as `mrustc` [2].

RustSmith supports the full range of Rust control flow constructs, including conditionals, loops and (potentially recursive) function calls, along with a wide range of data types such as dynamic and static arrays, tuples and structs. Currently out of scope features are concurrency, traits, generics, generators and unsafe code blocks.

We outline how to use RustSmith, discuss its design and implementation details, and showcase historic bugs found independently by RustSmith in various versions of `rustc` as well as previously unknown bugs in `mrustc`. We also present statement and mutation coverage results over the `rustc` optimiser codebase.

Availability and video demonstration. RustSmith is available as open source,[1] together with an artifact allowing `rustc` bugs and controlled experiments to be reproduced,[2] and a video walkthrough of the use of RustSmith.[3]

2 USING RUSTSMITH

The RustSmith repository provides binary releases and build instructions for the tool. Invoking the tool via the `rustsmith` command causes 100 random programs to be generated into an `outRust` directory, each with an accompanying text file containing the command-line arguments to be supplied to the generated program. RustSmith itself accepts command-line arguments to control the number of generated programs, output directory and random number generator seed.

RustSmith provides several generation modes that can be specified on the command line. The default mode controls the block sizes and mean recursive depths via parameters that we have manually tuned during the development of the tool, and results in programs with an average size $\approx 3,000$ lines.

3 DESIGN AND IMPLEMENTATION

Overall design. RustSmith works by recursively building a random abstract syntax tree (AST) conforming to the Rust grammar, which is then printed in textual form. Generation starts from the `main` function, and additional functions can be created during generation. To respect the Rust typing rules and semantics, generation is context-aware, e.g. a `break` is only generated inside a loop; and an `if` statement condition is restricted to type `bool`. To support this, RustSmith maintains a *symbol table*—akin to the symbol table of a compiler—that provides contextual information for each lexical scope of the program being generated, such as names and types of available variables, as well as globally-available information about constants, struct and function definitions.

Avoiding dangerous operations. Various "dangerous" operations are defined to yield runtime errors in Rust, including division by

[1] RustSmith on GitHub: https://github.com/rustsmith/rustsmith
[2] RustSmith artifact: https://doi.org/10.5281/zenodo.7940979
[3] Video walkthrough of RustSmith: https://youtu.be/ILdDLP2hH7I

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3604919

zero and accessing an array out-of-bounds. Integer overflow is specified as trapping in debug mode, but exhibiting wrap-around semantics in release mode. Careless generation of dangerous operations would lead to programs that abort with high probability, making them unlikely to be useful for finding compiler bugs, as well as making differential testing between release and debug modes a challenge. To address this, RustSmith follows Csmith and uses *wrappers* to guard potentially dangerous operations so that runtime errors are avoided (in both release and debug modes). For example, unsigned integer division is handled by a macro that yields zero if the denominator is zero, and the regular result of the division operation otherwise.

Borrowing. *Borrowing* and *lifetimes* (discussed below) are the key features of the Rust type system that allow memory safety (as well as freedom from data races) to be statically enforced. In Rust, objects are passed with *move* semantics by default: the source reference to the object becomes invalid, so that accesses via this reference are no longer allowed, and the receiving reference becomes the sole owner of the object. To support sharing of object references without object ownership changing, Rust allows objects to be *borrowed* through additional immutable or mutable references, denoted by `&` and `&mut` in type declarations, respectively. To ensure freedom from data races in a multi-threaded context, the owner of an object must only give out either (a) at most one *mutable* reference, or (b) any number of *immutable* references to the object. The Rust *borrow checker* checks that these rules are obeyed.

RustSmith does not generate concurrent programs, but *all* Rust programs must obey borrowing rules, so RustSmith must respect them. To this end, the RustSmith symbol table tracks the validity, mutability and borrowing states for reference variables and struct/tuple members. As an example, consider this Rust code:

```
fn main() {
  let mut a = 1i32;
  let b: &i32 = &a; // OK: multiple immutable borrows
  let c: &i32 = &a; // are allowed
  let d: &mut i32 = &mut a; // BAD: a is already borrowed
  *d = *b + *c;
}
```

RustSmith could generate the declarations `b` and `c`, which each borrow `a` immutably; the symbol table tracks the number of such borrows. But RustSmith would *not* generate the declaration `d`: knowing that `a` has already been borrowed at this program point, it is not legitimate for `a` to be borrowed mutably. Tracking validity information also allows RustSmith to ensure that `move`-semantics are respected, by avoiding uses of references after moves.

The restrictions that RustSmith places on generated program are slightly *stricter* than the actual borrowing rules of Rust, which only concern *live* references. It is acceptable for a mutably-borrowed object to be borrowed additionally (either mutably or immutably) as long as `rustc` can statically prove that at most one of the borrowing references will actually be used. In the above example, if `*d = *b + *c;` is changed to `*d = 1i32;` then `rustc` accepts the code despite the borrowing rules apparently being violated. This is because the immutable references `b` and `c` are no longer live: `rustc` can see that they will never be used, and thus there is no risk associated with `d` borrowing `a` immutably. Although this example is simple, mirroring exactly these kinds of edge-case programs that `rustc` accepts requires intricate data-flow analysis within RustSmith, which would be particularly difficult to achieve in the context of a program that is mid-way through being generated. Adopting slightly stricter rules suffices to ensure the generation of valid programs, at the cost of some loss in expressivity.

Lifetimes. Rust is not garbage-collected. Instead, every object has a unique owning reference, and each reference has a well-defined lifetime, related to its lexical scope. When the lifetime of a reference ends, the object that it refers to (if any) is automatically deallocated. The Rust type system has been designed to ensure that it is impossible for a reference to refer to an object whose owning reference's lifetime has ended. This achieves freedom from use-after-free errors that are typically a problem in non garbage-collected languages such as C and C++. Most object lifetimes in Rust are implicit—they are inferred by the compiler. However, explicit annotations are required in certain places, such as in reference members of structs. The compiler checks that the lifetime of a reference is *no longer than* the lifetime of the owner of the object that it points to.

To ensure that whenever a binding of a reference to an object is made, the object will outlive the reference, RustSmith exploits scoping information from the symbol table. Declarations at the top-level scope have the longest lifetimes, while declarations in a child scope have shorter lifetimes than declarations in enclosing scopes. Consider the following program fragment:

```
fn main() {
  let mut a = 100i32;
  let r: &mut i32;
   if (...) {
     r = &mut a; // OK: lifetimes of a and r match
  } else {
    let mut c = 10i32;
    r = &mut c; // BAD: r outlives c
  }
  *r = 42i32;
}
```

RustSmith could generate the assignment `r = &mut a;` because based on symbol table information it knows that `a` and `r` have matching lifetimes (as they are declared in the same scope). In contrast, the assignment `r = &mut c;` would *not* be generated, because RustSmith can deduce from the symbol table that `c` will not outlive `r` (as `c` is declared in a deeper scope than `r`).

When a struct is declared, RustSmith must equip its reference-typed members with appropriate lifetime annotations. To handle this, RustSmith creates structs "on demand", in that the struct definition is added the first time an object of the struct type is instantiated. For the struct definition, the current lifetimes of its reference-typed members will be used as lifetime parameters. An effect of this method of generation is that future instantiations of the struct will then only succeed with members having lifetime annotations identical to the first instance. This was simple to implement, but is conservative: it means that generated programs do not exercise the *lifetime coercion* mechanism of Rust, where values with longer lifetimes can be used in locations requiring a shorter lifetime.

Exercising the type inference engine. When generating a declaration statement (`let` *name* `:` *type* `=` *expr*), RustSmith internally determines a suitable type for the declaration, but randomly chooses

to omit it from the declaration, leaving it up to the compiler to infer the type and hence exercising its type inference engine.

Function inlining. RustSmith can randomly annotate generated functions with `#[inline(always)]` and `#[inline(never)]` to ensure that the compiler's inliner is exercised.

Inhibiting optimisations. RustSmith uses command-line arguments as a source of values that are unknown to the compiler. The use of such values ensures that an aggressive compiler cannot fully optimise a generated program to a trivial form.

Test-case reduction. Test cases produced by RustSmith can be large, so that an automated means of reducing them is required when they trigger a compiler bug. We find that the reduction tool C-Reduce [18], originally designed for C/C++, works well enough on RustSmith-generated programs to prove useful for this purpose.

4 BUG FINDING

During its development, we repeatedly used RustSmith in an uncontrolled fashion (and using its default generation mode) to test nightly builds of `rustc`. We also ran experiments against historic versions of `rustc`. The `rustc` version tags used were:

- 1.28 (the earliest version with all optimisation levels present)
- 1.40 (2 years old `rustc` version at time of development)
- 1.56 (`rustc` 2021 version)
- Latest (To always pick up latest stable version)
- Nightly (To pick up latest development builds)

This led to independent discovery of five historic bugs. We detail two of these, as well as one of two previously-unknown bugs affecting `mrustc` [2], an implementation of Rust in C++, found by RustSmith. We decided not to run RustSmith on historic versions of `mrustc` as `mrustc` is still a work in progress. The programs we show have been reduced from RustSmith-generated programs.

`rustc` bug: invalid opcode generated (affected v1.59–1.61). This program exhibits infinite recursion on `fun25`:

```
#[inline(never)]
fn fun25(var574: Box<i32>) -> ! { fun25(var574) }
fn main() {
  let var574 = Box::from(1745183449i32);
  fun25(var574);
}
```

Rust mandates that this should lead to a well-defined stack overflow error. Instead, when compiled with any optimisations enabled the program's execution leads to a runtime crash. We filed a bug report, but the issue turned out to have been independently reported two weeks earlier [13] and has now been fixed in `rustc` v1.62. Nevertheless, this demonstrates that RustSmith is capable of finding recent Rust miscompilations that developers care about fixing.

`rustc` bug: incorrect constant propagation (affected v1.45). This program exhibits different execution results when compiled with `rustc` v1.45 vs. v1.61:

```
fn main() {
  let mut var1: (bool, f64, i32) = (false, 0.5f64, 100i32);
  let var2: &mut bool = &mut var1.0;
  *var2 = true;
  let var3: (bool, f64, i32) = var1;
  if (var3.0) {
    let var4 = 10i32;
    println!("{:?}", ("var4", var4));
  } else {
    let var5 = 1i32;
    println!("{:?}", ("var5", var5));
  }
}
```

Due to an error in constant propagation, the v1.45-compiled version of the program incorrectly takes the `else` branch of the conditional. This problem was fixed in `rustc` in response to a bug report [4]. Again, it demonstrates RustSmith's ability to detect issues that developers deem worth fixing.

`mrustc` bug: crash on program featuring nested return. This program should compile without error and terminate normally:

```
fn fun1() -> () { return { return (); }; }
fn main() {}
```

However, with `mrustc` revision cd573b2, compilation crashes in `mir_builder.cpp`. The `mrustc` developers labelled our report of this issue [19] and another issue as "enhancement", reflecting the fact that `mrustc` is a self-professed "In-progress" project.

5 CONTROLLED EXPERIMENTS

We performed experiments assessing the coverage achieved by RustSmith on the `rustc` "mid-level IR" (MIR) optimiser—where the majority of Rust-specific optimisations are performed. For comparison, we also assessed coverage of the MIR optimiser achieved by the Rust *official optimisations test suite* (OOTS) within `tests/mir-opt`. Both code and mutation coverage experiments are performed on release v1.62.1 (Git hash `e092d0b6b`) of `rustc`.

Code coverage. We computed code coverage for the OOTS by running it against the coverage-instrumented compiler. To compute code coverage achieved by RustSmith, we generated 1,000 programs as a representative cohort (we repeated this three times and observed variance of 0.1% or below for both line and function coverage). See the `code-coverage` folder in our artifact for reproduction instructions. Unsurprisingly, the OOTS covered a large proportion of the MIR optimiser codebase, with overall line and function coverage of 82.12% and 72.78% respectively, compared to the RustSmith cohort's 51.59% and 54.06%. Although RustSmith was not written specifically to exercise these optimisations, RustSmith is able to achieve comparable coverage to the hand-written OOTS across many files, and achieved *better* coverage in multiple optimisations. For example, within `add_call_guards.rs`, RustSmith achieved 100.00% line coverage vs. OOTS's 55.56%; whilst within `nrvo.rs`, RustSmith achieved both higher function coverage (86.67% vs. 80.00%) and line coverage (94.12% vs. 93.38%).

Mutation coverage. To better compare the fault-finding ability of RustSmith vs. the OOTS, we turned to mutation coverage. While it would be possible in principle to perform exhaustive mutation analysis using the cargo-mutants project [17] we were concerned this would lead to an infeasibly large set of mutants when applied to the sizeable MIR optimiser code base. The mutation study can be reproduced following the instructions within the `mutation-coverage` folder of our accompanying artifact.

Based on previous evidence that miscompilation bugs often arise from incorrect optimisation preconditions [14, 15], we devised a

Table 1: RustSmith vs. OOTS on ability to kill mutants

	RustSmith killed	RustSmith survived	Total
OOTS killed	130	109	239
OOTS survived	3	112	115
Total	133	221	354

simple mutation operator that yielded a tractable set of mutants when applied over the MIR optimiser code base: negating the guard of an `if` statement. To avoid repeated recompilation of `rustc`, we devised a method to introduce mutants dynamically via an environment variable. Overall, mutations are applied to 354 distinct `if` statements across the MIR optimiser codebase. This provides a set of mutants that have been obtained systematically, and is small enough in number so that exhaustive analysis is feasible.

An input program *kills* a mutant if compiling the program using the mutated compiler produces *detectably different* results compared with the original compiler. Kills can arise due to compile-time panics or timeouts, or differences in generated object code. The latter may also lead to differences in results when the object code is executed (akin to miscompilation bugs).

To evaluate the mutation coverage of OOTS, we execute the full suite against each mutant, deeming a mutant killed if at least one test fails. The OOTS compares generated IR to expected IR, so that a test failures are either due to compiler panics or binary differences.

To evaluate the mutation coverage of RustSmith we performed a controlled experiment on an Ubuntu 18.04 machine with a 3.2GHz 8-core processor and 16 threads. RustSmith was applied to each mutant in turn, with a maximum time budget of 3 minutes to kill the mutant by repeatedly generating, compiling and executing programs. The 3-minute time limit was chosen to allow the experiment to finish within a reasonable timescale ($\approx$ 18 hours). The experiment would move on from a mutant either when this time budget was exhausted, or the mutant was found to be killable both via a compile panic *and* an execution mismatch. We were interested in both types of kill as neither subsumes the other in severity—compile panic is a compile-time symptom of the mutant, whereas an execution mismatch is a run-time symptom caused by the mutant.

Table 1 summarises the ability of the OOTS vs. RustSmith to kill mutants via this experiment. Three mutants were killed by RustSmith but survived the OOTS. The generated tests that kill these mutants could be refined into tests to be added to the OOTS.

6 RELATED WORK

A recent survey provides details of numerous compiler testing techniques [5]. RustSmith employs *differential testing* [16], which has had major impact in compiler testing through the Csmith [20] tool for C and various other "smiths" such as CLsmith [12] for OpenCL and Verismith [10] for Verilog. To our knowledge, RustSmith is the first differential compiler testing tool for Rust.

Several other successful approaches to randomised compiler testing are based on metamorphic testing [6], e.g. via *equivalence modulo inputs* testing [11], or semantics-preserving transformations [8]. We are not aware of any such methods targeting Rust.

Mutation based fuzzing tools such as AFL [9] and libFuzzer [1] have led to the discovery of numerous `rustc` bugs [3], but these are primarily front-end crashes triggered by invalid programs. In contrast, RustSmith allows end-to-end testing using *valid* programs.

In [7], the authors focus on `rustc`'s typechecker, by introducing mutations within the typing rules, and applying constraint logic programming to generate programs with subtle typing errors.

7 CONCLUSIONS

We have introduced RustSmith, a random generator of programs suitable for differential testing of Rust compilers. Our results show that RustSmith can find bugs in historic versions of `rustc`, find new bugs in `mrustc`, and achieve complementary statement and mutation coverage on the `rustc` optimiser codebase compared with the `rustc` official optimisation test suite. Directions for future work include extending RustSmith to support missing language features (e.g. traits, generics and unsafe code), and leveraging RustSmith as a source of new regression tests to fill test coverage gaps in `rustc`.

ACKNOWLEDGEMENTS

This works was supported by EPSRC grant EP/R006865/1.

REFERENCES

[1] [n. d.]. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html
[2] [n. d.]. Mutabah's Rust Compiler. https://github.com/thepowersgang/mrustc
[3] [n. d.]. Rust Fuzzing Trophy Case. https://github.com/rust-fuzz/trophy-case
[4] [n. d.]. The const propagator cannot trace references. https://github.com/rust-lang/rust/pull/73613
[5] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *Comput. Surveys* 53, 1 (2020), 1–36.
[6] T.Y. Chen, S.C. Cheung, and S.M. Yiu. 1998. *Metamorphic Testing: a New Approach for Generating Next Test Cases.* Technical Report HKUST-CS98-01. Department of Computer Science, The Hong Kong University of Science and Technology.
[7] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the Rust typechecker using CLP. In *ASE 2015*. IEEE, 482–493.
[8] Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.
[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.
[10] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *FPGA 2020*. ACM, 277–287. https://doi.org/10.1145/3373087.3375310
[11] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *PLDI 2014* (Edinburgh, United Kingdom). ACM, 216–226. https://doi.org/10.1145/2594291.2594334
[12] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. In *PLDI 2015*. ACM.
[13] Donough Liu. 2022. Infinite recursion optimized away with `opt-level=z`. https://github.com/rust-lang/rust/issues/97428
[14] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI 2021*. 65–79.
[15] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: How much does it matter? *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[16] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
[17] Martin Pool. 2023. cargo-mutants GitHub repository. https://github.com/sourcefrog/cargo-mutants
[18] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI 2012*. 335–346.
[19] Mayank Sharma. 2022. Bug thrown with nested return. https://github.com/thepowersgang/mrustc/issues/275
[20] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI 2011*. ACM, 283–294. https://doi.org/10.1145/1993498.1993532

Received 2023-05-18; accepted 2023-06-08