Rustlantis: Randomized Differential Testing of the Rust Compiler

QIAN WANG, ETH Zurich, Switzerland and Imperial College London, UK
RALF JUNG, ETH Zurich, Switzerland

Compilers are at the core of all computer architecture. Their middle-end and back-end are full of subtle code that is easy to get wrong. At the same time, the consequences of compiler bugs can be severe. Therefore, it is important that we develop techniques to increase our confidence in compiler correctness, and to help find the bugs that inevitably happen. One promising such technique that has successfully found many compiler bugs in the past is *randomized differential testing*, a fuzzing approach whereby the same program is executed with different compilers or different compiler settings to detect any unexpected differences in behavior.

We present *Rustlantis*, the first fuzzer for the Rust programming language that is able to find new correctness bugs in the official Rust compiler. To avoid having to deal with Rust's strict type and borrow checker, Rustlantis directly generates MIR, the central IR of the Rust compiler for optimizations. The program generation strategy of Rustlantis is a combination of statically tracking the state of the program, obscuring the program state for the compiler, and *decoy blocks* to lead optimizations astray. This has allowed us to identify 22 previously unknown bugs in the Rust compiler, most of which have been fixed.

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

Additional Key Words and Phrases: Differential fuzzing, Compiler testing, Rust

ACM Reference Format:
Qian Wang and Ralf Jung. 2024. Rustlantis: Randomized Differential Testing of the Rust Compiler. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 340 (October 2024), 27 pages. https://doi.org/10.1145/3689780

1 Introduction

Compilers are a critical piece of infrastructure. Whenever we are writing programs, debugging or testing them, even when doing verification, we usually work on the level of program source code, and we trust the compiler to correctly turn our intent into machine code. In the case of compilers for safe languages like Rust, Swift, or Go, we furthermore trust that compilation does not subvert the safety guarantees that the type checker has worked so hard to establish. However, modern compilers are also extremely complicated. For instance, the middle-end and back-end of the Rust compiler—the parts that work on the code after all type-checking is done—amount to around 100,000 lines of code (without comments and whitespace).[1] Once that code is done, further optimizations and machine code generation are performed by LLVM, which consists of multiple

[1]As reported by cloc `rustc_middle/src/mir rustc_const_eval rustc_mir_transform rustc_mir_dataflow rustc_codegen_ssa rustc_codegen_llvm rustc_codegen_cranelift` in the `compiler` directory on commit `ed7e35f34` of the Rust compiler (nightly-2024-07-06).

Authors' Contact Information: Qian Wang, andy.wang99@icloud.com, ETH Zurich, Switzerland and Imperial College London, UK; Ralf Jung, ralf.jung@inf.ethz.ch, ETH Zurich, Switzerland.

million lines of code. Clearly we cannot just expect all that code to be bug-free. We therefore need techniques that can help increase our confidence in the correctness of compilers.

The gold standard for this is full formal verification. However, while there have been landmark projects demonstrating the feasibility of end-to-end compiler verification, they are monumental efforts which are few and far between. Currently, there are verified compilers for two languages: C [17] and ML [14]. Clearly, it is not feasible to undertake such an endeavor for each language. And even if that were to happen, so far verified compilers are significantly lagging behind unverified compilers in terms of performance of the generated code.[2] We are quite far from being able to actually prove the correctness of the fastest production-grade compilers, or even their most important optimization passes. On top of this, that code is constantly evolving. Therefore, it is important that we consider alternative techniques that fall short of formal correctness proofs in their level of assurance, but are more easily applied to today's compilers.

One promising such technique is fuzzing. Fuzzing has been extremely effective at finding bugs in many different kinds of code [21], including compilers [7, 35, 15, 29, 24, 20]. One of the fuzzing techniques employed in this work is randomized differential testing [22]: the fuzzer generates random programs and then compiles them with various different compilers or different compiler flags (*e.g.,* with and without optimizations), and compares the results. If different results are produced, then one result must be wrong.[3]

However, so far no fuzzer has been developed that is able to effectively find new code generation correctness bugs in the Rust compiler. Rust is an interesting target for compiler correctness fuzzing, in the sense that it is both particularly worthwhile but also particularly challenging:

- Rust is a big language with a large number of control flow primitives. It has a complicated type system involving ownership types and borrowing (a form of region types). That makes it hard to randomly generate well-typed code.
- Rust programs can make use of `unsafe` operations, which can cause Undefined Behavior. Such programs can be compiled to arbitrary results without that being indicative of a compiler bug. It is quite common for Undefined Behavior to lead to different behavior with and without optimizations. Fuzzers hence have to be fairly sure that a program does not have Undefined Behavior before considering such behavior differences to be bugs. Furthermore, even safe Rust can exhibit non-determinism, which is in conflict with the differential testing methodology.
- Rust's primary targets are safety critical components, where compiler bugs could have a much bigger impact than, *e.g.,* in a UI frontend.
- Rust only has a single production-grade implementation, *i.e.,* a single compiler to worry about—improving the quality of that one compiler will benefit all Rust programmers. That said, this compiler has multiple backends: only the LLVM backend is considered stable, but there are experimental backends using Cranelift and GCC.

With this work, we introduce *Rustlantis*, the first fuzzer and random program generator for Rust that has been able to find new, previously unknown code generation correctness bugs. Overall, we have found 22 new bugs, most of which were quickly fixed by compiler developers. The key contributions of Rustlantis are as follows:

- Rustlantis generates a *control-flow graph* for a compiler IR, rather than generating regular Rust programs. This avoids the complexity of generating well-typed Rust code that passes

[2]According to the CompCert manual [18]: "Performance of the generated code is decent but not outstanding: on PowerPC, about 90% of the performance of GCC version 4 at optimization level 1." This refers to an ancient version of GCC, released around 20 years ago.

[3]This ignores issues such as non-determinism and Undefined Behavior; we will get back to those points later.

the borrow checker. It also means we cannot find bugs in the Rust frontend, but the frontend is already covered very well by other fuzzers [2, 13].

- Rustlantis uses a *place graph* to keep track of all reachable memory even through pointer indirections. This lets us generate code that we are sure is free of Undefined Behavior and non-determinism without adding local checks in the code that would make these properties obvious to the compiler. The generated code can even mix Rust's reference types and raw pointers, a situation that is prone to particularly subtle cases of Undefined Behavior [12, 31]. The place graph is interprocedural, so while Rustlantis knows everything about pointers passed as function arguments, intraprocedural analyses in the compiler do not. Some values are additionally obfuscated to prevent even interprocedural analyses from being able to reason about them.
- Rustlantis uses *decoy blocks* to generate code with a complex control flow that executes like straight-line code: the control flow graph can contain loops and call the same function in multiple places, but the actual execution of the generated program will not take any loop more than once and not call any function more than once to ensure the correctness of the data in the place graph.

While Rustlantis is only able to generate Rust programs, we believe that its fundamental approach is sufficiently general that it can be used to generate programs in control flow graph representation for other languages as well, such as LLVM IR or GCC GIMPLE.

To demonstrate the effectiveness of Ruslantis, we present the bugs we have found and reported: 8 bugs in rustc itself and 14 bugs in its code generation backends. In addition, we performed controlled experiments on specific Rustlantis and Rust toolchain versions, to measure bug detection rate and code coverage.

The rest of this paper is structured as follows: In §2, we provide the required background about differential testing and the Rust compiler architecture, in particular MIR. In §3, we describe the Rustlantis program generator. In §4, we explain our evaluation approach and result. Finally, we conclude with related and future work.

2 Background

Before explaining how Rustlantis works, we briefly discuss differential testing in general. We also explain the structure of MIR, the intermediate representation of the Rust compiler that Rustlantis works on.

2.1 Differential Testing

The general approach of differential testing [22] is to compile and run (or interpret) the same program with different compilers or interpreters and under different optimization settings—we call them *testing backends*. The compiler should not crash, and all execution results should be identical. If a compiler crashes or the execution results differ, then at least one testing backend has a bug, which can then be manually identified.

However, this only works if we are sure that the programs under consideration are *well-defined* (*i.e.,* they do not cause Undefined Behavior) and *deterministic*. These constraints are important because programs with Undefined Behavior may be optimized in arbitrary ways; there is no requirement of different backends producing consistent behavior. In contrast, non-deterministic programs are fundamentally well-defined, but they are allowed to exhibit multiple different behaviors at runtime, so different behaviors with different testing backends do not indicate a bug.

These are not just theoretical concerns; there are simple examples demonstrating how Undefined Behavior and non-determinism can lead to programs apparently changing their behavior when

```
fn ub() -> bool {
    let x: Option<bool> = unsafe { std::mem::transmute(13u8) };
    x.is_some()
}
fn nondet_behavior_ptr() -> bool {
    let x = 0i32;
    let y = 0i32;
    (&x as *const i32 as usize) + 4 == (&y as *const i32 as usize)
}
// Call with argument `0.0' to see the effect.
fn nondet_behavior_float(x: f32) -> bool {
    (x / 0.0).is_sign_positive()
}
```

Fig. 1. Examples of Rust programs that exhibit Undefined Behavior or non-determinism

comparing the result with and without optimizations. Those would lead to false positives during differential testing. We show three such examples in Figure 1.[4]

The first function, `ub`, is an example of Undefined Behavior. In an unoptimized build, this function returns `true`, but in an optimized build it crashes due to an "illegal instruction": the optimizer notices that this function cannot be reachable in a well-defined execution, and generates code that aborts the program.[5]

The other two functions demonstrate two sources of non-determinism. `nondet_behavior_ptr` compares the address of two local variables, checking whether they are exactly adjacent to each other (both variables have a size of 4 bytes). In an unoptimized build, it returns `true`, but in an optimized build it returns `false`.[6] `nondet_behavior_float` shows that memory addresses are not the only source of non-determinism in Rust: performing a floating-point division of `0.0` by `0.0` produces a `NaN` result. Even NaN values carry a sign, but Rust considers the sign to be picked non-deterministically. In an unoptimized build, this function returns `false`, but in an optimized build it returns `true`.[7]

2.2 Mid-level Intermediate Representation (MIR)

MIR is the last intermediate language of the Rust compiler before lowering to LLVM IR (or Cranelift, in the case of the Cranelift backend). It is the IR used for borrow-checking and also for various optimization passes. That makes it by far the most relevant IR of the compiler for our purposes. Conveniently, the Rust compiler has a feature called "custom MIR" that enables writing test cases that directly produce a given MIR (rather than writing regular Rust which gets compiled to MIR).

[4]We have tried these examples with Rust 1.76.0 on an x86-64 Linux machine. Results can vary with different compiler versions and different targets. Unoptimized builds use rustc without any flags; optimized builds additionally set the `-O` flag.
[5]The reason why this function has Undefined Behavior is not very important, so we delegate the explanation to a footnote. 13 is not a valid representation for the type `Option<bool>`, so this unchecked typecast ("transmute" in Rust terms) violates language assumptions. With optimizations, the compiler realizes that this cast is never allowed and replaces the entire function by an "unreachable" trap.
[6]The exact addresses that local variables are placed at are not specified by Rust. In an unoptimized build, it happens to be the case that `y` is stored just after `x` in memory. In an optimized build, both variables get removed entirely and the compiler just pretends that it picked a stack layout where the variables happened to not sit next to each other.
[7]In an unoptimized build, the division is performed by the x86 CPU at runtime, which generates `NaN` with negative sign. In an optimized build, the division is performed by a soft-float library at compile-time, and the soft-float library generates `NaN` with positive sign.

Custom MIR can be injected at any stage in the MIR pipeline. Rustlantis uses this feature to be able to entirely bypass the Rust frontend and inject MIR between borrow-checking and optimizations.

MIR does not have a concept of "safe" vs. "unsafe" code, so the programs generated by Rustlantis would in general require unsafe Rust to be written in the surface language. In fact, many of the generated programs cannot be directly expressed in surface Rust at all; however, they might arise from regular Rust programs by applying MIR transformations. Characterizing the exact subset of MIR that can arise in today's Rust compiler is non-trivial, and it is not worth the effort: to be prepared for future compiler changes, MIR passes are designed to be able to deal with *all* well-formed MIR. Therefore, any bug in a Rustlantis-generated program is considered a bug in the compiler, even if today it would not be possible to hit that bug by writing regular (safe or unsafe) Rust code.

High-level Rust operations such as iterators and pattern matching are reduced to sequences of low-level operations in MIR, so a deep understanding of Rust is not required to read MIR code. However, MIR uses the same types as surface Rust, so we briefly discuss the particularities of Rust types that are relevant here. First of all, Rust distinguishes between *references* and *(raw) pointers*. References are safe to use and fully tracked by the type system; raw pointers are almost the same as pointers in C and can only be used in unsafe code, because the type system does not keep track of whether they are valid or not. Both of these come in a *mutable* and an *immutable* variant: `&mut T` and `&T` for mutable and immutable references; `*mut T` and `*const T` for mutable and immutable pointers. The same MIR syntax is used to dereference all of them. Secondly, Rust has `enum` types that correspond to algebraic datatypes from functional programming languages. For instance, an `enum OptionInt { Some(i32), None }` represents a value that is *either* some integer (of type `i32`), or no value at all. Here, `Some` and `None` are called the *variants* of the enum.

With that out of the way, we can talk about the structure of MIR itself. MIR is a control-flow graph (CFG) language where each function body contains one or more *basic blocks*. Each basic block consists of any number of *statements* followed by a *terminator*. Statements within a basic block are to be executed top-down with no branching or function calls, and they never diverge (but they can cause Undefined Behavior). The terminator leads to one or more successor basic blocks. The program's control flow, such as if-statements, loops, and function calls, are represented by different terminators. In addition to basic blocks, MIR function bodies also contain declarations of local variables and their types. The grammar of the subset of MIR Rustlantis can generate is listed in Figure 2.

Statements. Rustlantis generates only a single kind of statement: assignments. (MIR itself supports more statements, but they are either purely administrative or used extremely rarely.) Assignments store an "rvalue" into a place.

Rvalues (named after the fact that they appear on the **r**ight-hand side of assignments) provide the basic computational primitives of MIR: arithmetic (unary and binary operators, as well as "checked" operators which indicate whether overflow occurred), creating a reference or raw pointer to a place, as well as basic datatype constructors. The operands of all these operations can either be constant literals, or they can be loaded from a place. The special `Move` operand type is only relevant for function calls; we will explain it later in §3.2.

Places (also known as "lvalues" in C as they appear on the **l**eft-hand side of assignments) represent a location in memory. Syntactically, a place expression consists of a local variable combined with zero or more *projections*. Rustlantis can generate 4 types of projections:

- Tuple or struct field: `a.0`, `a.foo`

⟨*Function*⟩ ::=
    ‘fn’ ⟨*ident*⟩‘(’ (⟨*ident*⟩‘:’ ⟨*type*⟩,)* ‘) ->’ ⟨*type*⟩ ‘{’ (⟨*Declaration*⟩;)* ⟨*BasicBlock*⟩$^+$ ‘}’

⟨*Declaration*⟩ ::= ‘let’ ⟨*ident*⟩ ‘:’ ⟨*type*⟩

⟨*BasicBlock*⟩ ::= (⟨*bb*⟩ ‘=’)$^?$ ‘{’ (⟨*Statement*⟩;)$^+$ ⟨*Terminator*⟩ ‘}’

⟨*bb*⟩ ::= ⟨*ident*⟩

⟨*Terminator*⟩ ::= ‘Goto(’ ⟨*bb*⟩ ‘)’
 | ‘Return()’
 | ‘Call(’ ⟨*Place*⟩ ‘=’ ⟨*ident*⟩ ‘(’ (⟨*Operand*⟩,)* ‘)’,
        ‘ReturnTo(’ ⟨*bb*⟩ ‘)’, ‘UnwindUnreachable()‘ ‘)’
 | ‘match’ ⟨*ident*⟩ ‘{’ (⟨*literal*⟩ ‘=>’ ⟨*bb*⟩,)* ‘_ =>’ ⟨*bb*⟩ ‘}’

⟨*Statement*⟩ ::= ⟨*Place*⟩ ‘=’ ⟨*Rvalue*⟩

⟨*Rvalue*⟩ ::= ⟨*Operand*⟩
 | ⟨*UnOp*⟩ ⟨*Operand*⟩
 | ⟨*Operand*⟩ ⟨*BinOp*⟩ ⟨*Operand*⟩
 | ⟨*Operand*⟩ ‘as’ ⟨*ty*⟩
 | ‘Checked(’ ⟨*Operand*⟩, ⟨*BinOp*⟩, ⟨*Operand*⟩ ‘)’
 | ‘&’ ⟨*Place*⟩ (create immutable reference)
 | ‘&mut’ ⟨*Place*⟩ (create mutable reference)
 | ‘&raw const’ ⟨*Place*⟩ (create immutable pointer)
 | ‘&raw mut’ ⟨*Place*⟩ (create mutable pointer)
 | ‘(’ (⟨*Operand*⟩,)$^+$ ‘)’ (create tuple)
 | ‘[’ (⟨*Operand*⟩,)$^+$ ‘]’ (create array)
 | ⟨*type*⟩ ‘{’ (⟨*ident*⟩‘:’ ⟨*Operand*⟩,)$^+$ ‘}’ (create struct)

⟨*Operand*⟩ ::= ⟨*Place*⟩ | ‘Move(’ ⟨*Place*⟩ ‘)’ | ⟨*literal*⟩

⟨*UnOp*⟩ ::= ‘!’ | ‘-’

⟨*BinOp*⟩ ::= ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘^’ | ‘&’ | ‘|’ | ‘<<’ | ‘>>’ | ‘==’ | ‘<’ | ‘<=’ | ‘!=’ | ‘>=’ | ‘>’

⟨*Place*⟩ ::= ⟨*ident*⟩
 | ⟨*Place*⟩ ‘.’ ⟨*index*⟩ (tuple field)
 | ⟨*Place*⟩ ‘.’ ⟨*ident*⟩ (struct field)
 | ⟨*Place*⟩ ‘[’ ⟨*index*⟩ ‘]’ (array element)
 | ‘Field(Variant(’ ⟨*Place*⟩, ⟨*index*⟩ ‘)’, ⟨*index*⟩ ‘)’ (enum variant field)
 | ‘*(’ ⟨*Place*⟩ ‘)’ (pointer dereference)

Fig. 2. MIR grammar generated by Rustlantis

- Field of an enum variant: `Field(Variant(a, 2), 3)` projects to the 2nd variant of place `a`, and then the third field inside that variant (this operation does not exist in surface Rust, it is generated as part of compiling pattern matching down to MIR)
- Array element: `a[0]`, `a[i]`
- Reference and pointer dereference: `*a`

Arbitrarily many (or zero) projections can be chained together in a place expression. However, later stages of the Rust compiler (including the one where Rustlantis will inject its generated MIR) require that dereference is only used as the first projection. For instance, `*(a.foo)` is not allowed, as the dereference occurs after a field projection.

Due to pointers, syntactically identical place expressions may evaluate to different places at different points in the program. For instance, consider:

```
1  let x: i32; let y: i32;
2  let a: *mut i32;
3  {
4      a = &raw mut x;
5      *a = 42;
6      a = &raw mut y;
7      *a = 42;
8      Return()
9  }
```

`*a` on line 5 is syntactically identical to `*a` on line 7, but the former refers to `x`, whereas the latter refers to `y`. Conversely, pointers may alias, so there can be syntactically distinct place expressions referring to the same location.

Terminators. Rustlantis supports the following four terminators:

- **`Goto`**: Unconditionally enter another basic block.
- **`match`**: Conditional jump based on the value of a local variable. If nothing matches, enter the fallthrough basic block. (This corresponds to a `switch` in C.)
- **`Call`**: Call another function, then enter the target basic block after it has returned.
- **`Return`**: Copy the value of the return slot (a dedicated local variable) to the `return_place` in the corresponding **`Call`** terminator, then return from the current function and enter the target basic block.

The most notable omission compared to the full grammar of MIR is `Drop`, the terminator used to call a destructor. Destructor calls usually get automatically inserted by the Rust compiler whenever a variable goes out of scope. However, these calls behave just like regular function calls, so from the perspective of fuzzing code generation they are not very interesting.

3 Rustlantis

Rustlantis can generate programs from the grammar shown above in Figure 2. We support all primitive integer and floating point types, **`bool`**, **`char`**, arrays, tuples, raw pointers, shared and mutable references, structs, and enums. On top of all the usual primitive arithmetical and logical operators, Rustlantis supports pointer arithmetic and `transmute` (an unsafe type cast).

Rustlantis generates single-file programs containing a `main` function which calls other generated functions. The source program can either be compiled into an executable and then executed natively, or be interpreted by Miri [26], a Rust interpreter intended for Undefined Behavior detection.

Note that Rustlantis only generates *monomorphic* code, *i.e.,* code without generics or traits. This suffices to test code generation because the Rust compiler anyway monomorphizes code before handing it to the backend: at this point, all types are known and all function calls are resolved to the concrete instance that will be invoked.[8] That said, MIR optimizations are applied before

[8]Rust also supports dynamic dispatch via `dyn Trait` types, which is not supported by Rustlantis. While dynamic dispatch is important in some scenarios, the vast majority of Rust code uses static dispatch.

monomorphization, so for a future extension of Rustlantis it could be interesting to consider generating generic code.

By construction, programs generated by Rustlantis are terminating, deterministic, and free from Undefined Behavior (UB). The specification of Undefined Behavior in Rust is still incomplete; when there are gaps in the specification, we use the definition implemented by Miri with the Tree Borrows [31] aliasing model. Miri is able to detect all Undefined Behavior that the Rust compiler exploits for optimizations, so this guarantees that a difference in output between different testing backends always indicates a bug in at least one of the backends or a bug in Rustlantis.

The rest of this Chapter describes the generation process in detail.

3.1 Statements and Declarations

Rustlantis' generation process is similar to an interpreter, but instead of executing code, it is producing code in execution order. Within a basic block, Rustlantis generates statements or declarations top-down, one at a time. It maintains a cursor which specifies the current function and current basic block (represented as `fnX:bbY`). The next statement/terminator to be generated will be added at the end of that basic block.

To produce an executable program, Rustlantis also generates a `main` function that calls the initial function `fn0` with a list of literal arguments. While Rustlantis knows the values chosen for these arguments, we want them to be "hidden" from the compiler and not used by optimizations. For this purpose, Rust provides a `black_box` function, so even if `fn0` gets inlined, the values chosen for its arguments cannot be used by optimizations.

Generation starts at `fn0:bb0`. At each step, Rustlantis decides whether it should generate a statement or a new local variable declaration.

Each statement is generated by randomly traversing its grammar, as visualized in Figure 3.

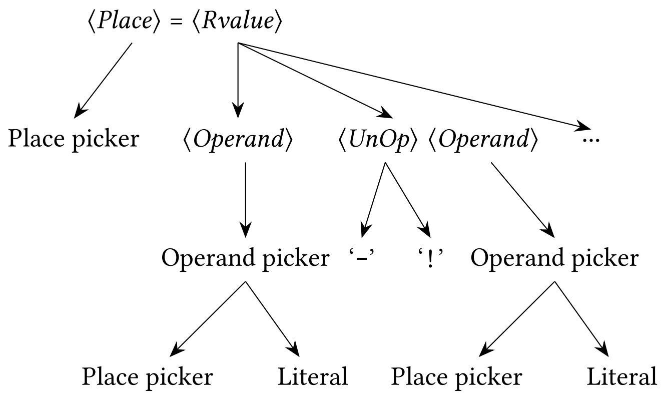


Fig. 3. The decision tree for generating an assignment.

For an assignment statement, Rustlantis needs to choose two sub-components: a place and an rvalue. The place is the left-hand side (LHS) of the assignment. A dedicated function will provide a list of candidate places, which is described below in §3.2.

Assuming for now that we have chosen a place from the candidate list, the next step is to call a function that generates an rvalue which has the same type as the chosen LHS. If this is not possible (there is no expressible place in the current function with that type), then Rustlantis removes the chosen LHS place from the candidate list, picks another place and then tries again. This repeats

```
1  fn fn1() { mir!(
2    let _1: (i32, bool);
3    let _2: *const bool;
4    {
5      _1 = (42, true);
6      _2 = &raw const _1.1;
7      // <- Current Cursor
8    }
9  )}
```

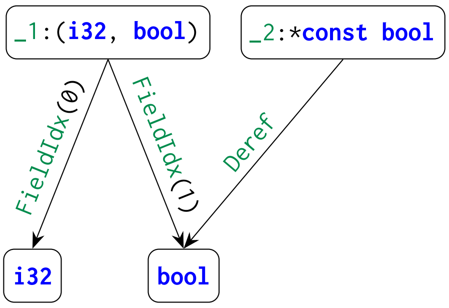


Fig. 4. Place graph example

until either a valid pair of place and rvalue is found, or the candidate list becomes empty and an error is returned.

When an operand is required as a part of an rvalue variant, we first try to pick a place as above. If this fails *and* the required operand type is expressible as a literal, we generate a random literal.

After recursively trying all options, the root that generates the assign statement may run out of place candidates. In this case, a new local variable is declared in the current function, making it more likely that we will find a suitable place for the next statement.

To generate statements that are interesting, the choices at each syntactical level have weights attached. For instance, we do not want to generate a standalone Operand as Rvalue too often, but we do want to generate `&raw` often as raw pointers are complex to optimize. We determined these weights by how complex we estimate the operations are for the compiler to optimize, and hard-coded them.

3.2 Keeping Track of Places

We get back to the question of how to generate random places. Recall that a key concept of places in Rust are *projections*, which map one place to another: from a local variable to its field to a field of its field. This naturally gives rise to a graph structure, where nodes are places, and edges are projections. This graph contains trees rooted at each local variable (formed of the projections that access fields and index arrays), but it's not a forest since dereference projections can point from one tree to another, to reflect where the pointer currently points to.

In Rustlantis, the *place graph* is responsible for keeping track of all places in the program, globally. Figure 4 shows a representation of the place graph when the cursor is on line 7.

When a local variable declaration is generated, the variable is added to the place graph, along with all the places reachable via field/index projections. Dereference projections for places of pointer type are only added when a statement assigning to the pointer is generated. This means we know at all times where an initialized pointer is pointing to. If a pointer is overwritten, the old dereference edge is removed and a new one is added targeting the new pointee.

Note the place graph isn't a fully accurate memory model for Rust: unlike C and C++, Rust does not have "typed memory". This means that a local variable does not contain a tree of values reflecting the structure of its type, it just contains an untyped list of bytes. The same memory location can be referred to by different place expressions of different types (assuming the alignment constraint is satisfied). Nonetheless, the place graph is a valid, albeit restricted, approximation to Rust's full memory model.

Each node in the place graph also keeps track of some information about the runtime state of the place, such as whether it is initialized. In case a literal assignment to it is generated, the value of the literal is saved as the *known value* of the place. The value stops being known when it is overwritten by a non-literal, *e.g.,* the result of an arithmetic operation.[9] When an enum value is assigned to a place, the variant index is also saved to the node as a known value. The information in a node can be propagated to another node when an operation that copies between places is generated.

Generating place candidates. Whenever the program generator needs a place, we gather all local variables and perform a depth-first search through the place graph starting from each of them. This gives us the full list of expressible places. Then we filter out the places that could produce an ill-formed program or trigger UB if used in the current context.

For instance, when we are selecting the divisor of a `Div` binary operation, we can restrict the selection to places that have the same type as the dividend, are initialized, and have a non-zero known literal value. The filtering guarantees that all place candidates are remaining choices that will not result in a compile error or UB.

When Rustlantis enters a new function, all the local variables in the previous function (the caller) are no longer accessible. To keep track of which local is currently accessible, we track a *place stack* alongside the place graph, with one frame for each ongoing function call.

Each frame in that stack contains a list of nodes in the place graph which are locals declared in the function. Nodes are added to the last frame whenever a local declaration is generated, and place candidate searches start from the locals in the current frame.

Finally, there is subtle Undefined Behavior around function calls that we need to be aware of: the return place designated by the caller (*i.e.,* where the return value will be stored) must not be accessed while the function runs. It is not possible to write Rust code that does this, since Rust always uses a fresh local variable as the return place for each function call. However, in MIR it is possible to write code like `Call(_1 = f(&raw mut _1), ...)`, and such MIR might arise from regular Rust code after applying some MIR optimizations. To allow more efficient handling of return values, Rust declares it Undefined Behavior for `_1` to be accessed while `f` runs, so reading or writing that raw pointer passed as first argument would be Undefined Behavior.[10] Rustlantis hence considers `_1` to be *protected* during the execution of `f`. This is tracked in the place stack.

A similar situation arises with “call-by-move” function arguments, as in `f(Move(_3))`. Such arguments may use an optimized calling convention that avoids copies, at the cost of making it Undefined Behavior to access the original place while the function runs. Hence, `_3` would also be considered *protected* in this example.

An example demonstrating both protected arguments and return values is shown in Figure 5. The cursor is in `fn2`. While executing this function, the local variables `_1` and `_3` in `fn1` are protected, since they are used as return place and call-by-move function argument, respectively. `fn2` would have the chance to access both of these places via the raw pointers `a` and `c`, respectively; doing that would cause Undefined Behavior. Since the place graph tracks where these pointers are pointing, Rustlantis can detect this situation and avoid generating such accesses.

After a function has returned, all its local variables are deallocated. Accessing a deallocated place is UB, so we add information about whether a place has been deallocated to the place graph. Deallocated places can never be chosen. This prevents UB if a function returns a pointer to a local variable.

[9]It is theoretically possible to compute all operations and know the value of all places. However, this would require a full interpreter, which would be significantly more implementation effort in the fuzzer.

[10]The underlying reason for this is that the Rust compiler may store the callee’s return local directly in the place designated by the caller. The return local can be read or written by the callee like any other local, so any use of that memory for other purposes is forbidden while the function executes.

```
fn fn1() { mir!(
  let _1: i32;
  let _2: *mut i32;
  let _3: Foo;
  let _4: *mut Foo;
  {
    _2 = &raw mut _1;
    _3 = Foo { /* ... */ };
    _4 = &raw mut _3;
    Call(_1 = fn2(_2, Move(_3), _4), ReturnTo(bb1), UnwindUnreachable())
  }
  bb1 = {
    // Return target
  }
)}

fn fn2(
    a: *mut i32,
    b: Foo,
    c: *mut Foo
) -> i32 { mir!(
  {
    // <- Current Cursor
  }
)}
```

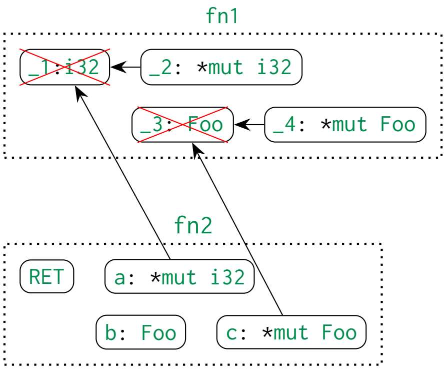


Fig. 5. Place graph with multiple functions

3.3 Terminators

Once a random amount of statements in a basic block is reached (chosen at the beginning of generating a block), a terminator is generated, and the generation of the basic block is complete. Rustlantis then resumes generation in the basic block that execution will jump to. This ensures that we always follow the same path as program execution, so the information in the place graph is always accurate.

`Goto`. When a `Goto` terminator is selected, an empty basic block is added to the current function and the cursor is set to it to resume generation there.

`SwitchInt`. This is the most complex terminator, as there are multiple potential successors. Our strategy is as follows (illustrated in Figure 6):

(i) First, we pick a place with a known value as the subject to match on.
(ii) Then, we generate a list of *decoy* arms for values that we know are not the current value of the subject. The target of a decoy arm can be a random existing basic block like `bb1`, or new empty basic blocks like `bb3` and `bb4`.
(iii) If we have added new decoy basic blocks, we fill them with an identical copy of a random existing basic block, including statements and the terminator. (If there are no existing ones to choose from, we add to it the `Return` terminator and no statements.) This is guaranteed to be syntactically well-formed, as all identifiers mentioned in existing basic blocks have already

(i)

```
bb1 = {
  x = 42;
  /*...*/
}
bb2 = {
  match x {
    ...
  }
}
```

(ii)

```
bb1 = {
  x = 42;
  /*...*/
}
bb2 = {
  match x {
    0 => bb1,
    1 => bb3,
    _ => bb4,
  }
}
bb3 = {}
bb4 = {}
```

(iii)

```
bb1 = {
  x = 42;
  /*...*/
}
bb2 = {
  match x {
    0 => bb1,
    1 => bb3,
    _ => bb4,
  }
}
bb3 = {
  /* copy of bb1 */
}
bb4 = {
  /* copy of bb1 */
}
```

(iv)

```
bb1 = {
  x = 42;
  /*...*/
}
bb2 = {
  match x {
    0 => bb1,
    1 => bb3,
    42 => bb5,
    _ => bb4,
  }
}
bb3 = {
  /* copy of bb1 */
}
bb4 = {
  /* copy of bb1 */
}
bb5 = {
  /* <- New Cursor */
}
```

Fig. 6. Steps of generating a **`SwitchInt`**

been declared. The semantic effect of the content is irrelevant, as these basic blocks will never be executed at runtime.

(iv) Finally, we add the real target basic block for the known value of the subject. We move the cursor to it and resume generation.

This approach guarantees that all statements are executed at most once, and all functions are entered exactly once. Our generation order is in lockstep with the real execution order of the program, thereby guaranteeing that all UB can be prevented using the accurate information in place graph.

Although Rustlantis knows the precise value of the subject, the compiler cannot always determine this value. For instance, the subject value may come from dereferencing a pointer which is a function parameter. The compiler would have to find the value of its pointee from the callers – potentially multiple levels up the call chain. With decoy blocks, there can be multiple call sites for each function, so the compiler cannot easily guarantee that the pointer parameter always points to the same value.

This strategy allows us to produce a CFG similar to the ones that can be produced from surface Rust programs containing loops, if-else/match statements, and break statements. The resulting CFG can be quite complex and exercise edge cases in the compiler. However, it is guaranteed to be always reducible.

`Call`. When a **`Call`** terminator is selected, we pick a random place as the return place, and a random amount of operands as arguments. Then we add an empty basic block to the current function as the return target and add a new function using the types of the selected return place and arguments as the signature. Finally, we set our cursor to the first basic block of this new function.

We need to know the return target basic block to know where to resume once we return from the new function, so we need to maintain a *return target stack* as a part of Rustlantis' global state. The return target is pushed onto the stack whenever a `Call` terminator is generated.

As discussed above, we also push a new frame to the stack that's tracked with the place graph, to record the return place node and all `Move` arguments generated in the terminator.

`Return`. When a `Return` terminator is selected, we must first check the place graph to see if the return slot (the dedicated local variable `RET`) is initialized, as it is UB to return from a function with an uninitialized return slot, even if the return value is never used in the caller. If `RET` is uninitialized, we cannot produce a `Return` terminator, so we select another terminator instead.

Once the function returns, we

(1) Copy the content of `RET`'s node in the place graph node to the return place.
(2) Pop the last frame off the call stack.
(3) Mark all places contained in locals in the popped frame as deallocated in the place graph (*i.e.,* all their fields and elements—nodes behind pointer indirections remain live, of course).
(4) Set our cursor to the return target basic block in the caller and resume generation.

3.4 Representing Memory Layout

The `transmute` intrinsic is Rust's form of an unsafe cast (like `reinterpret_cast` in C++ or `Obj.magic` in OCaml). We would like to generate calls to `transmute` to make sure the compiler does not do any type-based tracking when that would be incorrect: it is completely okay to write to some location using one type and then read using a different type, as long as the value stored at that location is compatible with both types. (This is different from C, where type-based aliasing restrictions disallow such "type punning".)

However, generating correct calls to `transmute` is non-trivial. Types in Rust may contain gaps between their fields (called *padding*), and it is Undefined Behavior to read padding. As a consequence, when transmuting between values of different types, Rustlantis needs to reason about the layout of source and destination to ensure that we do not read from padding bytes.

However, the default type representation `repr(Rust)` does not guarantee fixed memory layouts for tuples and structs. This means that we cannot rely on these types to have any specific size, or that their elements reside at specific offsets. Figure 7 shows some possible memory layouts of a type declared as `struct MyStruct(i8, i16)`. Although a specific version of the Rust compiler may use a fixed layout computation algorithm, this cannot be relied on. Indeed, the Rust compiler has a flag `-Z randomize-layout` to make the layout of each type in the default representation unpredictable.

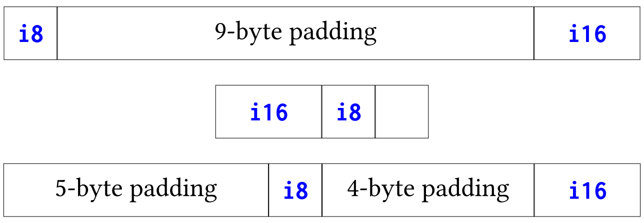


Fig. 7. Some possible memory layouts of `struct MyStruct(i8, i16)`

As we are only dealing with types using the default representation, any types that *may* contain padding have an indeterminable size and element layout and therefore cannot be transmuted.

However, there are types with guaranteed sizes and no paddings. These include primitive integers, floating points, **bool**, **char**, pointers, and arrays of these types (including arrays of arrays).

We say places of these types fit into a *run*, which represents a contiguous region of memory of a known size and without padding bytes. The place graph tracks for each node whether it is part of a run. In the case of arrays, both the array itself and its elements are associated with the same run. Figure 8 shows a representation of the place graph with runs.

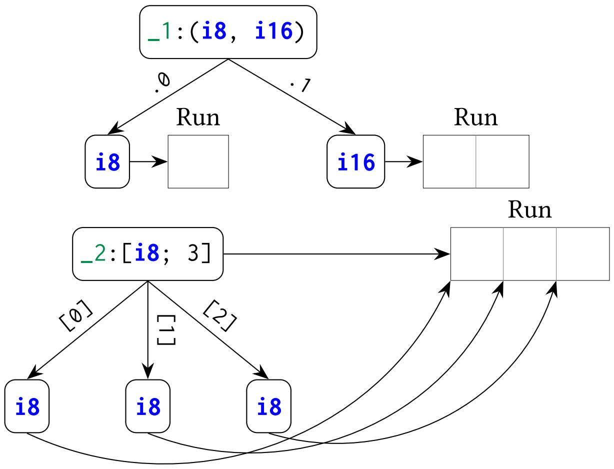


Fig. 8. Places pointing to runs

When we are picking the argument to a `transmute`, we restrict the place candidates to ones that have an associated Run of the size of the return place where the result will be stored. This ensures that there is no padding in the input.

`transmute` has an additional constraint regarding value validity: the bit pattern in the source must represent a valid value for the destination type. Of the types Rustlantis generates, only **bool** and **char** have restrictions for what their valid bit patterns are. It is possible to uphold this validity constraint using the known value information, but we have not yet implemented this fine-grained filtering. For now, we simply prevent transmutations to **bool** and **char** types.

3.5 Pointers and References

All relevant information about raw pointers is tracked in our place graph: we know where each pointer points to, so we can make sure that we only store pointers to allocated memory, and only load data of the right type.

However, references in Rust are subject to further constraints. While a reference is live, it cannot dangle (point to deallocated places), and its pointee must be initialized and contain a valid value for the type. Beyond that, Rust imposes *aliasing restrictions* on references: mutable references must be unique and shared references must be read-only. Aliasing models such as Stacked Borrows [12] or Tree Borrows [31] define what exactly is required to ensure these properties. Roughly speaking, the idea is that for shared references, the pointee cannot be written (through *any* pointer) while the reference is live. For live &**mut** references, the pointee can be neither read nor written, except through the &**mut** reference. A reference is considered *live* between its creation and last use. Additionally, a reference-typed function argument is *live* for the whole duration of the function call.

To avoid Undefined Behavior from violating these constraints, Rustlantis performs borrow tracking during program generation with a simplified version of Stacked Borrows [12]. A borrow stack is associated with every byte in every run. When a pointer or reference is created with &, a *tag* is created to represent this pointer. This tag, along with the type of borrow (raw pointer, shared reference, or mutable reference) is placed onto the borrow stacks of all bytes in all runs that have been referenced. This is illustrated by Figure 9.

```rust
let x: u8 = 0;
let a = &raw const x; // Tag 1
let b = &x; // Tag 2
let c = &(*a); // Tag 3
```

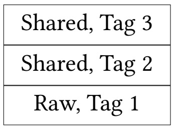


Fig. 9. Representation of a borrow stack

A tag in a borrow stack is *potentially borrowed from* all tags below it. This is imprecise, as a borrow created later in program order is not necessarily derived from an existing pointer, but it allows us to track borrows with only a stack instead of a tree.

The borrow stack behaves according to these rules:

When reading, writing, or creating a reference from a pointer, we check

- that the tag exists on the borrow stack (ensuring the pointer is valid), and
- that the tag is not below any mutable reference (ensuring we are not invalidating any mutable reference).

When writing through a pointer, we additionally check

- that the tag is not above a shared borrow (ensuring this pointer is not potentially derived from a shared borrow, which would make it illegal to write through this pointer), and
- that the tag is not below any tag associated with a reference-typed function argument (ensuring we do not invalidate references that must stay live for the duration of a call).

When a place is written, we pop off all tags above the tag used to perform the write.

When a reference-typed value is produced, we check that the pointee isn't deallocated and is initialized.

Together, these rules guarantee that Rustlantis does not generate any programs that trigger Undefined Behavior involving references.

3.6 Pointer Arithmetic

Besides `transmute`, the other key unsafe operation Rustlantis generates is pointer arithmetic. Specifically, we generate calls to the `arith_offset` intrinsic, which takes a pointer and an `isize` offset as arguments. This intrinsic is never UB to call (its non-intrinsic counterpart, `wrapping_offset`, is a safe function). However, it may offset a pointer outside the bounds of its allocation, in which case that pointer would be UB to dereference.

To track where the pointer points, we choose some places with known values as the offset argument. However, we do not impose a restriction on its value. We keep track of the offset amount from its original pointee in the place graph node of each pointer-typed place, accumulating the offset on each call to `arith_offset`.

Whenever we are walking the place graph to find place candidates, we only visit dereference projections of pointers whose effective offset value is zero. The offset value is zero if the pointer has never been offset, or if all its offsets add up to zero and therefore “roundtripped". This guarantees that we can dereference the pointer without UB.

With information on known values and runs, it is possible to offset pointers to array elements to other elements within the same array and dereference them while having a non-zero offset. This is an interesting option for future work.

3.7 Picking "interesting" Places

If place candidates are picked uniformly at random, the resulting program will contain a large number of unused assignments. In many cases, the LHS of an assignment will never be used or is overwritten before it is used. Unused assignments are trivially optimized out early in the compilation pipeline, never exercising the more interesting parts of the compiler.

Additionally, we assume that a complex series of operations is more likely to trigger miscompilations. Therefore, when picking which places should affect the output of the program (see §3.9), we would like to pick the results of complex operations. Printing a place containing a literal that was assigned immediately before is not very interesting.

We introduce a measure to represent the *data complexity* of each place, indicating the amount and complexity of operations which took place to result in the value in a place. The complexity is tracked for each node in the place graph, and this complexity is propagated between places through dataflow. On each assignment, the complexity value of the rvalue is calculated and becomes the new complexity of the LHS place.

The complexity value of each rvalue is calculated as follows:

- Literal operands have a complexity value of 1.
- Place-based operands take the complexity value from its underlying place.
- & takes the complexity value from its pointee place.
- Other rvalues' complexity value is the sum of all its operands'.

The complexity of a place with multiple fields (structs and tuples) is the maximum complexity of its field places.

The complexity of a place is used as a weight in place selection: while picking a place to be read (such as for an operand), we favor places containing complex data to propagate the complexity.

This means that the complexity of places can accumulate in a positive feedback loop. We cap the complexity of each place at 100 to prevent some places from having an exceptionally high complexity and getting chosen every time.

During place candidate selection, the weight of each place is further augmented: we always favor places containing a dereference projection, and especially favor places dereferencing a pointer that has been "roundtripped" by pointer offsetting. This makes it likely that we will later dereference an arithmetically modified pointer, which we expected to be prone to miscompilations (such as Bug #9 in Figure 10).

We perform additional weight augmentations depending on the selection context: if the place is used as a function argument, we favor places that are pointers (especially those that have been offset), or places with known values. If the place is selected for the LHS of an assignment, we ignore its complexity, but favor places that are uninitialized.

3.8 Ensuring Determinism

The biggest difficulty in Rustlantis is generating code that is free of Undefined Behavior. We have explained how every single step in program generation ensures that all preconditions for the generated instructions are met, and therefore the resulting program is well-defined. However, that is not sufficient: we must also avoid introducing non-determinism. Even well-defined Rust programs can be non-deterministic. If our program is non-deterministic, running it with different backends can legitimately produce different results, so non-deterministic programs are not suited for differential testing.

The main source of non-determinism is of course concurrency, but we are only generating sequential programs, so this is of no concern. Besides concurrency, Rust has two sources of non-determinism that affect Rustlantis: the addresses that local variables are stored at, and the exact bit-pattern returned by a floating-point operation that produces a NaN ("not a number") value. We saw examples of both in Figure 1.

To ensure that this non-determinism cannot affect program behavior, Rustlantis generates programs in a way that the non-determinism is contained to values of pointer or float type. Specifically, we say that any type that contains a pointer or float is *non-deterministic*, and all remaining types are *deterministic*. Concretely, this means we have to avoid the following operations that could leak data from non-deterministic types to deterministic types:

- pointer-to-int casts and transmutes,
- pointer comparisons, and
- floating-to-int transmutes and operations that expose the sign of a floating point value.

Note that float-to-int *casts* are fine as those will reliably produce 0 for any NaN value, so the non-determinism in the NaN bit-pattern cannot affect the resulting integer. Control flow can only be affected by integers and Booleans (those are the only valid operands for a MIR **`match`**), which are deterministic, so there is no risk of indirect leakage.

3.9 Producing Observable Output

A generated program must expose its runtime state to the outside world. Otherwise, we have no way of knowing whether the testing backends deviate from each other.

Before we generate a **`Return`** terminator, we choose some local variables with deterministic type weighed by their complexity values and call a gadget function, `dump_var`, to print them to the standard output. The gadget takes the function name, and the names and values of four variables and print them out. It is called multiple times if more than 4 variables are chosen to be dumped, and a variable of the unit type `()` is supplied if the number of chosen variables is not a multiple of 4 (the arity 4 is experimentally chosen to be the fastest). We do not print all local variables, as this will often reduce the amount of possible optimizations, thus reducing the chance of encountering a miscompilation.

The printed string makes it immediately obvious which variable in which function has a different value in different testing backends. However, this causes a significant slow down in differential testing. We use an alternative version of `dump_var` which hashes the variables and prints the final hash value once at the end of the program.

To preserve both speed and bug visibility, we observe the fact that the vast majority of programs do not trigger miscompilations, and will produce the same output whichever `dump_var` we use, miscompilations from the small minority of programs should be detectable by both versions of `dump_var`. We can have the best of both worlds by using the fast `dump_var` usually, and when a different hash is detected, we generate the program with the slow, debug `dump_var` and run differential testing again to see the difference with better visibility.

3.10 Summary

In summary, Rustlantis' tracks the following state during program generation: the so-far generated program, the cursor, the place graph, the place stack, and the return target stack.

In the place graph, nodes represent places and edges represent projections. Each node additionally keeps track of: its value complexity, a tag (for pointers/references), information about the value stored in that place, and a set of runs (tracking information about the memory layout).

This lets us basically interpret the program as it is being generated, allowing Rustlantis to generate deterministic, UB-free, and diverse MIR programs. These programs are effective at triggering bugs in the Rust compiler, which will be demonstrated in the next section.

4 Evaluation

4.1 Bug Finding

We used the following differential testing backends in our fuzzing campaign:

(1) Ahead-of-time compilation with MIR optimizations and LLVM optimizations & backend.
(2) Ahead-of-time compilation with only LLVM optimizations & backend.
(3) Ahead-of-time compilation with only Cranelift optimizations & backend.
(4) Interpretation with Miri.

All of these backends are managed in the Rust language repository and distributed with the “nightly” version of the Rust toolchain.

LLVM is the default backend of the Rust compiler. It comes with an extensive optimization suite and a codegen backend for many targets. On top of that, the Rust compiler also has its own optimization suite that acts on MIR.

Cranelift [1] is a machine code generator developed by the Bytecode Alliance and is itself implemented in Rust. rustc can use Cranelift as the code generation backend as an alternative to LLVM. It is much younger than LLVM and far less widely used, therefore we hypothesized that it may contain more bugs due to its immaturity.

Rust also has a backend using GCC for code generation. However, that backend is incomplete and doesn’t support all Rust features yet; at the time we started the project, it was not able to compile many of the programs we generated, so we did not include it in our fuzzing campaign.

Miri [26] is a Rust interpreter which executes MIR. It can detect and report UB encountered at runtime. If Miri reports UB on a Rustlantis-generated program, then there is a bug in Rustlantis and the execution results of the same program from other testing backends must be discarded. This has occurred during development, but we have fuzzed tens of millions of programs on the most recent versions of Rustlantis and no UB has been reported. The interpreter used by Miri is shared with rustc where it is also used as part of MIR optimizations (specifically, constant propagation and global value numbering), so a bug in Miri can still indicate a potential miscompilation.

The combination of testing backends allows us to detect bugs in rustc’s MIR optimizations, LLVM, Cranelift, and Miri/const-prop: A bug in rustc can cause backend 1 to be different. A bug in LLVM can cause backends 1 and 2 to be different. A bug in Cranelift can cause backend 3 to be different. A bug in Miri/const-prop can cause backends 1 and 4 to be different.

Fuzzing with Rustlantis is trivially parallelizable as multiple instances can be run simultaneously with different seeds using GNU Parallel [30]. It is therefore well-suited for HPC clusters. We have been running a fuzzing campaign with Rustlantis on an x86_64-based Linux cluster, and very occasionally on an Apple Silicon Mac. In total we performed 10 CPU years of fuzzing (two months worth of wall-clock time) against nightly Rust versions ranging from 2023-03-23 (`1459b3128`) to 2024-03-03 (`516b6162a`).

We found 24 unique confirmed bugs, 22 of which have not been previously known. All of these are correctness bugs or crashes; we have not discovered any hangs. All but two of them have been fixed at the time of writing. Figure 10 shows all newly discovered bugs in detail as well as a breakdown of these bugs by their origin.

#	Report	Type	Origin	Description	Fixed
1	rust#110902	Crash	rustc	Assertion failure in `RenameReturnPlace`	Y
2	rust#110947	Miscompilation	rustc	`ConstProp` propagates over mutating borrows	Y
3	rust#111426	Crash	rustc	`ReferencePropagation` prevents partial initialization	Y
4	rust#111502	Miscompilation	rustc	`*const T` in function parameters annotated with `readonly`	Y
5	rust#112061 llvm#63019	Miscompilation	LLVM	Aliasing analysis merges loads from different offsets	Y
6	llvm#63013	Crash	LLVM	Phi nodes assumed to be non-empty	Y
7	llvm#63033	Crash	LLVM	Assertion failure in `RegisterCoalescer`	N
8	rust#112170 llvm#63055	Miscompilation	LLVM	Constant folding produces invalid boolean values	Y
9	rust#112526 llvm#63266	Miscompilation	LLVM	Aliasing analysis is broken for overflowing pointer offsets	Y
10	rust#112548 llvm#97147	Miscompilation	LLVM	Lowering of multiplication instruction causes overflow on AArch64	Y
11	rust#112767 llvm#63430	Miscompilation	LLVM	Copy elision corrupts stack arguments with two parts	Y
12	llvm#63475	Miscompilation	LLVM	Copy elision reads stack arguments from the wrong offsets	Y
13	rust#113407	Miscompilation	rustc	Subnormal `f64` to `f32` cast is wrong	Y
14	llvm#64897	Miscompilation	LLVM	Incorrect size merging for `MustAlias` sets	Y
15	rust#117355	Crash	rustc	MIR inlining inserts statements at the wrong place	Y
16	rust#118328	Miscompilation	rustc	`ConstProp` propagates over assignment of unknown values	Y
17	llvm#74890	Miscompilation	LLVM	Bad `undef` and `poison` handling in `InstCombine`	Y
18	cg_clif#1455 cranelift#7865	Crash	Cranelift	Multiplication with out of bounds shift panics	Y
19	rust#120613	Miscompilation	rustc	GVN merges moved function arguments	Y
20	llvm#82884	Miscompilation	LLVM	`GVNPass` forgets to remove poison generating flags	Y
21	cg_clif#1460 cranelift#7999	Miscompilation	Cranelift	Misoptimization of `imul` with `ireduce`	Y
22	rust#121996 llvm#84025	Miscompilation	LLVM	`InstCombine` calculates wrong `insertelement` instructions	Y

	Miscompilation	Crash
rustc	5	3
LLVM	10	2
Cranelift	1	1

Fig. 10. All previously-unknown bugs discovered by Rustlantis, and a breakdown by component.

16 out of the 22 previously-unknown bugs are miscompilations, the most serious type of compiler bug. LLVM contained the most bugs, which is unsurprising as it is the most complicated part of the compilation pipeline by far. Despite being new, Cranelift held up well with only two bugs discovered by Rustlantis, thus rejecting our hypothesis that it might be less mature than the more established LLVM backend.

4.2 Bug Examples

We discuss some of the previously-unknown bugs discovered by Rustlantis.

Bug #2: Constant propagation across mutating pointer. This is a miscompilation in rustc's MIR optimization. The reduced MIR was rewritten into this surface Rust function:

```
1  pub fn fn0() -> bool {
2      let mut pair = (1, false);
3      let ptr = &raw mut pair.1;
4      pair = (1, false);
5      unsafe {
6          *ptr = true;
7      }
8      let ret = !pair.1;
9      return ret;
10 }
```

`fn0` should return `true`, but instead it returned `false`. This is due to Rust's `ConstProp` MIR optimization incorrectly propagating the constant value of `pair.1` (`false`) from line 4 to line 8 despite the mutation through a pointer on line 6.

Bug #8: `undef` *in booleans.* This is an LLVM bug with a 3-line minimal reproducing example:

```
define i64 @test(double %arg) {
  %cmp = fcmp une double 0x7FF8000000000000, %arg
  %ext = zext i1 %cmp to i64
  ret i64 %ext
}
```

`0x7FF8000000000000` is a NaN (not-a-number) bit pattern under the IEEE-754 double-precision floating-point format. The instruction `fcmp une` checks if the operands are not equal. The value of `%cmp` should be `true` as `NaN` does not equate anything (not even itself), therefore the function should return `1`.

However, LLVM 16.0.4 generates code that always returns `255`. This is due to the `fcmp une` instruction being incorrectly folded into an `undef` constant, which turn into any bit pattern.

Bug #21: Combination of multiplication and integer conversion. This is the first and only miscompilation Rustlantis has found in Cranelift. The reproducing example was manually reduced to a small function in Cranelift IR:

```
function u0:11(i8) -> i8 system_v {
block0(v0: i8):
    v1 = uextend.i64 v0 ; Zero-extend v0 to i64
    v2 = imul_imm v1, 256 ; Multiply by 256
    v3 = ireduce.i8 v2 ; Convert back to i8 by discarding the most significant bits
    return v3
}
```

This function should always return `0`: after a 64-bit integer is multiplied by 256 on line 4, the least significant 8 bits are all 0, so the conversion on line 5 should always result in zero. However, due an unexpected interaction between two rewriting rules, the optimized IR returns the function argument unchanged.

4.3 Time to First Bug

Our fuzzing campaign was performed with evolving versions of Rustlantis and rustc. The metrics gathered from this campaign have too many variables to allow any meaningful conclusions. To gather a more meaningful estimate of how long it takes Rustlantis to find a bug, we conducted a controlled experiment: we generated 100,000 random programs using the latest version of Rustlantis and performed differential testing on 11 historical Rust nightly toolchain versions. We gathered the number of bug-triggering programs for each Rust version, and calculated the statistically expected time to first bug (measured in number of programs). The results are presented in Figure 11.

Rustlantis' ability to trigger bugs fluctuates significantly across compiler versions. This is likely due to different bugs being fixed and introduced in different Rust versions. In earlier versions of rustc, we often get hundreds of examples failing due to the same root cause. In later versions, bugs become harder to hit; it is quite common to only see the first bug well after 100,000 programs have been tested, and some bugs are exhibited only once. For comparison, the Csmith authors [35] report a bug rate of 0.0024% on the latest LLVM version (2.8) they have tested, which means an expected time-to-first-bug of about 41,000 programs.

4.4 Code Coverage

Here, we compare branch coverage and line coverage metrics between Rustlantis and RustSmith [28], which to our knowledge is the only other Rust fuzzer aimed at finding code generation correctness bugs. In terms of bug-finding effectiveness, Rustlantis outperforms RustSmith: RustSmith rediscovered 5 already known bugs but was not able to find any new bugs in rustc.

Code coverage is often used to measure a fuzzer's effectiveness at finding bugs [3]. We generated 200 random programs with both Rustlantis and RustSmith using the `2024-03-23` nightly version of the Rust compiler on x86_64, and measured the cumulative branch and line coverage from compiling these programs (Figure 12). We are most interested in code coverage in the `rustc_mir_transform` crate in Rust, and `lib/Analysis`, `lib/Transform`, `lib/Codegen`, and `lib/Target` directories in LLVM, as these represent the root cause of almost all bugs discovered by Rustlantis, and are the most correctness-critical yet error-prone parts of the compiler.

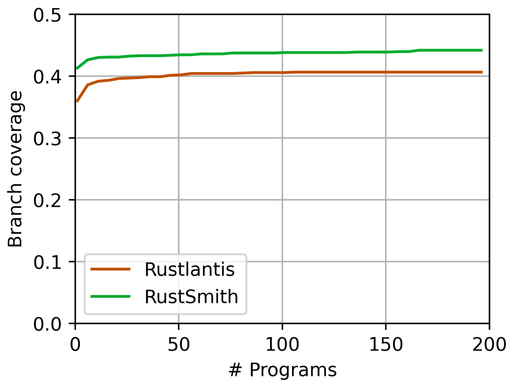
Despite being a far more effective fuzzer in terms of bugs found, Rustlantis exercises a slightly smaller proportion of Rust and LLVM's codebase. This confirms the observation [35, 21] that code coverage is not a good metric for comparing fuzzers for compilers, likely because the metric is unable to capture the subtle invariants and relations that actually determine whether an optimization is working correctly. The results also show that overall coverage is rather low, in particular in LLVM, probably because of optimization passes that are not enabled by rustc.

4.5 Resource Usage

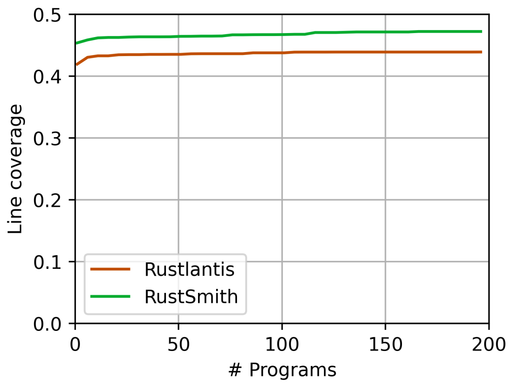
On a single core of AMD EPYC™ 9654, the median time for Rustlantis to generate a program is 0.20 second, and it takes further 2.79 seconds to perform differential testing against the four testing backends sequentially. 95% of the programs are generated by Rustlantis using less than 24 KB of memory, and tested using less than 1.3 GB of memory.

Toolchain	Miscompilations	Crashes	Bug rate (%)	Exp. first bug
2023-05-01	2,122	619	2.741	36
2023-06-01	0	2	0.002	50,000
2023-07-01	0	2	0.002	50,000
2023-08-01	0	2	0.002	50,000
2023-09-01	5,315	2	5.317	19
2023-10-01	0	2	0.002	50,000
2023-11-01	3	99	0.102	980
2023-12-01	0	2	0.002	50,000
2024-01-01	0	2	0.002	50,000
2024-02-01	0	0	0.000	> 100,000
2024-03-01	0	0	0.000	> 100,000

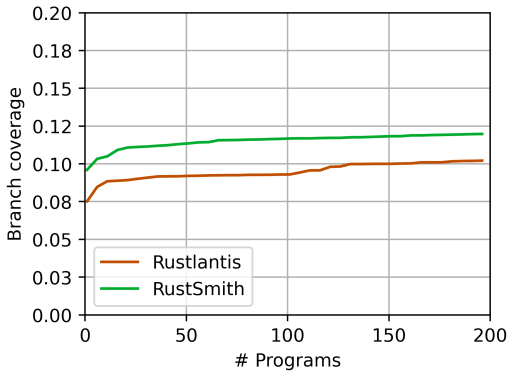
Fig. 11. # of bug-triggering programs in historical Rust versions, out of the same 100,000 generated programs



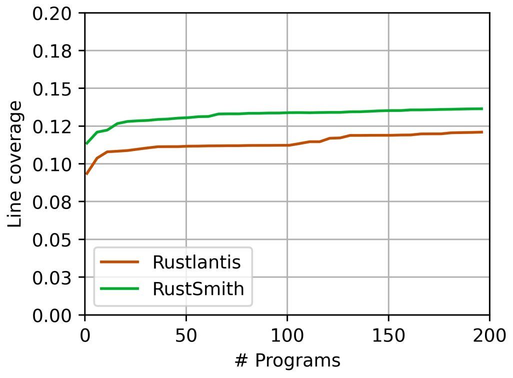
(a) Branch coverage of `rustc_mir_transform`



(b) Line coverage of `rustc_mir_transform`



(c) Branch coverage of LLVM backend



(d) Line coverage of LLVM backend

Fig. 12. Coverage of the relevant parts of the Rust compiler and LLVM

5 Related Work

The recent survey by Chen et al. [4] gives a good overview of current techniques used for compiler testing. Here we compare directly with the most relevant pieces of prior work: program generators for Rust and other languages, as well as the alternative approach of mutation-based testing.

Rust program generation. RustSmith [28] was already mentioned above as the only fuzzer we are aware of that aims to detect miscompilation bugs in rustc. The main difference in our approach is that RustSmith generates safe Rust, as opposed to Rustlantis which generates MIR that generally can only be obtained by writing unsafe Rust. In fact, Rustlantis often generates MIR that does not correspond to any Rust program—but it is still crucial to ensure the correct treatment of such MIR, since it could be the result of earlier optimization passes. RustSmith was not able to find any previously-unknown bugs in rustc, but it was able to re-discover five bugs that had already been discovered before, and it found new bugs in mrustc, an alternative Rust compiler written in C++. Rustlantis cannot fuzz mrustc as mrustc is not based on MIR.

Pearce [25] defines a core calculus for Rust-style type and borrow checking, and uses that to fuzz-test the Rust frontend. They found at least one bug in the borrow checker; however, code generation bugs are out of scope for this work.

AFL and *libFuzzer* are general-purpose fuzzing libraries that have been used to generate random Rust programs [2]. These programs are not necessarily well-formed. This approach has been effective at discovering panic, infinite loop, and out-of-memory bugs, thus covering a disjoint part of the compiler compared to Rustlantis.

Dewey et al. [6] use *Constraint Logic Programming* to generate well-typed Rust programs. Their approach was able to discover precision, soundness, and consistency bugs in Rust's typechecker, but it did not probe the codegen backend.

Generating programs for other languages. Csmith [35] generates deterministic and UB-free C programs and has found numerous bugs in the optimization pipeline and backend of multiple C compilers. It has been the original inspiration for Rustlantis. Csmith uses a form of static dataflow analysis combined with safe wrappers around unsafe operations (such as signed integer arithmetic, where overflow would lead to UB in C) to ensure that the program does not cause Undefined Behavior. This analysis is comparable to Rustlantis' place graph, but less precise. On the one hand, this means there are programs that Rustlantis generates where a static analysis may lose track of what exactly the set of aliases of some pointer is. Rustlantis also avoids the need for wrapper functions and can directly invoke unsafe operations while being sure that their preconditions are met. On the other hand, this means Csmith can generate programs with loops that are executed more than once, and functions that are called more than once.

CsmithEdge [8] builds on Csmith but instead of excluding UB by construction during program generation, they use dynamic UB detection techniques to determine whether the program has UB once it is generated. For Rustlantis, this means we could generate, *e.g.,* divisions or `transmute` calls without worrying about whether the result is well-defined or not. We could use Miri for UB detection. However, we would want to avoid generating too many programs that have UB, as running Miri takes some time and those programs cannot be used to find correctness bugs.

Nagai et al. [23] generate random C programs that are free of Undefined Behavior. However, it focuses on finding bugs in arithmetic optimizations.

YARPGen [20] improves on Csmith by avoiding the need for wrappers to ensure safety of signed integer arithmetic. The approach is similar to Rustlantis: for *mixed* variables, *i.e.,* those that are both read and written by the generated program, the tool tracks their exact current value. Mixed variables are also never used in loops. Rustlantis, in contrast, can generate control-flow that looks like a loop to the compiler, and still track the contents of all variables as the tool knows that the loop is only taken once. In the future it would be interesting to explore adding some of YARPGen's loop generation features to Rustlantis, and thus generate loop with dedicated induction and reduction variables that otherwise work on data which does not change during the loop execution.

Hansson [11] explore random code generation for a language even later in the compilation pipeline than MIR or LLVM IR: they generate random programs in LLVM MIR, the Machine Intermediate Representation that is used for instruction selection. They were able to find several crash bugs, but the programs they generate may contain Undefined Behavior and non-determinism, which makes it impossible to find code generation correctness bugs. Their approach to control-flow graph generation is a bit different from ours: where our CFG is reducible by construction (despite being more general than what can be directly obtained with loops and conditionals), they generate a random CFG and then make it reducible by applying suitable graph transformations.

Ofenbeck et al. [24] generate random programs in a user-defined DSL to find bugs in the DSL compiler. However, they assume that all generated program are well-defined and deterministic; as such, this approach is not suited for languages like C or MIR where program generation has to be careful to avoid Undefined Behavior and non-determinism.

Recent work has attempted to use machine learning techniques to generate random programs for compiler testing [5, 19, 34]. While these program generators are getting better and better at producing syntactically valid programs, they are still nowhere close to ensuring properties such as well-defined behavior and determinism, and as such are not suited for finding correctness bugs.

Mutation-based testing. Instead of generating random programs entirely from scratch, mutation-based approaches work by changing existing test cases. This turns out to be extremely effective.

Icemaker [13] is a tool for finding Rust compiler bugs that runs continuously. It often finds new crash bugs within hours of them landing in the compiler, and is responsible for a significant fraction of the recent crash reports for rustc. However, it cannot find code generation correctness bugs.

Equivalence Modulo Inputs [15, 16, 29] refers to a class of fuzzers that mutate an existing program in a way that does not change the observable behavior under a given input, such as by applying arithmetic identities or mutating dead code. This may trigger different optimizations in live code paths, and has found a large number of miscompilation bugs in GCC and clang/LLVM—more than any of the generation-based approaches, as far as we know. (And yet, generation-based approaches still find new bugs, even after equivalence modulo inputs reached saturation.) Rustlantis' decoy basic blocks can be seen as a lightweight form of EMI, where the fuzzer knows that the code is dead and can hence produce arbitrary syntactically valid code.

6 Conclusion and Future Work

We have presented *Rustlantis*, a new program generator for fuzzing the Rust compiler. Rustlantis can generate a wide range of programs, including those that use unsafe features such as raw pointers, mixing pointers and references, pointer arithmetic, and `transmute`. The combination of directly building a *control-flow graph*, tracking knowledge about the current program state in a *place graph*, and adding *decoy blocks* to confuse the optimizer enabled Rustlantis to find 22 previously unknown bugs in rustc: 6 crashes and 16 incorrect compilation results. This makes Rustlantis the first fuzzer that was able to find any new code generation correctness bugs in the official Rust compiler.

Despite this successful result, there are various ways in which Rustlantis could be improved.

6.1 Generating a Larger Variety of Rust Programs

Rustlantis does not generate some common constructs in Rust, such as: unions, recursive types, heap-allocated (`Box`) types, zero-sized types, manually dropped types, non-`#[repr(Rust)]` types, and many more. Supporting these constructs could exercise more of the compiler and thus find more bugs. In particular, `repr(C)` types would be interesting as they have a guaranteed layout, so they could be used to generate more complicated invocations of `transmute`.

Speaking of unsafe operations, as already mentioned the support for pointer arithmetic in Rustlantis is limited in that pointers are only accessed if the cumulative offset is zero. Support for doing pointer arithmetic to other array elements or fields of `repr(C)` structs would have a greater chance of confusing the compiler. Likewise, it would be interesting to be able to generate `offset` operations, which is a form of pointer arithmetic that requires the pointer to stay in-bounds of the allocation and thus lets the compiler deduce further information.

Another key unsafe operation is a pointer type cast, where memory is accessed under a different type than the one with which it was originally initialized. This is comparable to `transmute`, but does not perform a copy—the same memory can now be accessed under two different types, which would be challenging to track in the place graph, but also has good potential of triggering further compiler bugs.

Finally, it would be interesting to extend Rustlantis with support for generating generic code. That would let us test whether MIR optimizations (which are applied before monomorphization) handle such code correctly.

6.2 Better Exploring the Space of Possible Programs

Rustlantis implicitly defines a space of possible programs and a decision tree to randomly pick a sample from that space. Currently, this strategy is rather naive; there is no attempt to cover the space of programs uniformly, or to use feedback from the testing process to guide the exploration to other parts of the space (*e.g.*, coverage-guided fuzzing). It would be interesting to explore such options in the future.

Another possibility is to employ Swarm Testing [10], where a set or “swarm” of samples is generated, each of which focuses on a different part of the program space.

6.3 Automated Program Reduction

Rustlantis-generated programs are too long to be directly used in a bug report. As a result, if a generated program triggers a bug, it must be reduced to a far smaller *minimal complete verifiable example* (MCVE). This is also sometimes called *test-case shrinking*. We have primarily reduced the test cases manually, with the help of a very simple script that comments out one line at a time and checks whether the bug is still reproducible. This script is naive and inefficient. When the bug is in LLVM and we were able to isolate a reproduction in LLVM IR form, we used `llvm-reduce` [9] to minimize the IR. However, this tool can sometimes introduce Undefined Behavior during reduction and thus invalidate the test program. Many sophisticated techniques have been proposed in prior work [27, 4]; adopting them to MIR programs could make the production of an MCVE a lot simpler.

Data-Availability Statement

Rustlantis’ source code is maintained on GitHub [32]. A Docker image containing Rustlantis and the scripts used for §4 is permanently archived on Zenodo [33].

Acknowledgments

We thank the Rust compiler developers and in particular Nikita Popov for being extremely quick at fixing the bugs we found, which was crucial for the fuzzing campaign to continue and discover other bugs. The fuzzing campaign was primarily performed on the Euler cluster of ETH Zurich.

References

[1] Bytecode Alliance. 2018. Cranelift Code Generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift

[2] Rust Fuzzing Authority. 2017. rust-fuzz. https://github.com/rust-fuzz/trophy-case

[3] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22).* http://seclab.cs.stonybrook.edu/lszekeres/Papers/ICSE22.pdf

[4] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (2 2020), 36 pages. https://doi.org/10.1145/3363562

[5] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 95–105. https://doi.org/10.1145/3213846.3213848

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

[7] Eric Eide and John Regehr. 2008. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, Luca de Alfaro and Jens Palsberg (Eds.). ACM, 255–264. https://doi.org/10.1145/1450058.1450093

[8] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empir. Softw. Eng.* 27, 6 (2022), 129. https://doi.org/10.1007/S10664-022-10146-1

[9] Diego Treviño Ferrer. 2019. LLVM-Reduce for testcase reduction. (2019). https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech22 2019 LLVM Developers' Meeting.

[10] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *ISSTA*. ACM, 78–88.

[11] Bevin Hansson. 2015. *Random Testing of Code Generation in Compilers.* Master's thesis. Royal Institute of Technology, Stockholm. https://robcasloz.github.io/teaching/BevinHansson_2015.pdf

[12] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2019. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL, Article 41 (dec 2019), 32 pages. https://doi.org/10.1145/3371109

[13] Matthias Krüger. 2020. icemaker. https://github.com/matthiaskrgr/icemaker

[14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

[15] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14).* Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2594291.2594334

[16] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 386–399. https://doi.org/10.1145/2814270.2814319

[17] Xavier Leroy. 2009. A formally verified compiler back-end. *JAR* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[18] Xavier Leroy. 2023. The CompCert C verified compiler - Documentation and user's manual. https://compcert.org/man/manual.pdf

[19] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.* AAAI Press, 1044–1051. https://doi.org/10.1609/AAAI.V33I01.33011044

[20] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (nov 2020), 25 pages. https://doi.org/10.1145/3428264

[21] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: how much does it matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (oct 2019), 29 pages. https://doi.org/10.1145/3360581

[22] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[23] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Trans. Syst. LSI Des. Methodol.* 7 (2014), 91–100. https://doi.org/10.2197/IPSJTSLDM.7.91

[24] Georg Ofenbeck, Tiark Rompf, and Markus Püschel. 2016. RandIR: differential testing for embedded compilers. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche (Eds.). ACM, 21–30. https://doi.org/10.1145/2998392.2998397

[25] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 3:1–3:73. https://doi.org/10.1145/3443420

[26] The Rust Project. [n. d.]. Miri. https://github.com/rust-lang/miri.

[27] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 335–346. https://doi.org/10.1145/2254064.2254104

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

[29] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 849–863. https://doi.org/10.1145/2983990.2984038

[30] Ole Tange. 2023. *GNU Parallel 20230822 ('Chandrayaan')*. https://doi.org/10.5281/zenodo.8278274 GNU Parallel is a general parallelizer to run multiple serial command line programs in parallel without changing them..

[31] Neven Villani. 2023. *Tree Borrows*. Master's thesis. ENS Paris-Saclay. https://github.com/Vanille-N/tree-borrows/blob/eeb44c2509a6fa3f6e55f4bd75f5fd416a576676/half/main.pdf

[32] Andy Wang. 2023. Rustlantis. https://github.com/cbeuw/rustlantis

[33] Andy Wang and Ralf Jung. 2024. Reproduction Image for Article 'Rustlantis: Randomized Differential Testing of the Rust Compiler'. https://doi.org/10.5281/zenodo.12670660

[34] Haoran Xu, Yongjun Wang, Shuhui Fan, Peidai Xie, and Aizhi Liu. 2020. DSmith: Compiler Fuzzing through Generative Deep Learning Model with Attention. In *2020 International Joint Conference on Neural Networks, IJCNN 2020, Glasgow, United Kingdom, July 19-24, 2020*. IEEE, 1–9. https://doi.org/10.1109/IJCNN48605.2020.9206911

[35] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. *SIGPLAN Not.* 46, 6 (jun 2011), 283–294. https://doi.org/10.1145/1993316.1993532

Received 2024-04-06; accepted 2024-08-18