Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Semantic fuzzing of the Rust compiler and interpreter

Master's Thesis

Qian (Andy) Wang

31 July 2023

Advisor: Prof. Dr. Ralf Jung
Department of Computer Science, ETH Zürich

https://t.me/learningnets

Abstract

This project introduces *Rustlantis*, a novel fuzzer capable of generating programs in Rust's Mid-level Intermediate Representation that are deterministic and free from Undefined Behaviour. It has uncovered 13 previously-unknown bugs in the Rust compiler and LLVM which has caused miscompilations as well as crashes.

https://t.me/learningnets

Acknowledgements

I would like to thank my supervisor, Prof. Ralf Jung, for his expert guidance on Rust's semantics, which has been crucial for the success of this project.

I would also like to thank Nikita Popov for his rapid response, investigation and fixes to bug reports I have filed during the course of this project.

https://t.me/learningnets

Contents

Contents **iii**

1 Motivation **1**

2 Background **3**
2.1 Fuzzing 3
2.2 Differential testing 3
2.3 The Rust compiler 4
2.4 Rust fuzzing 5
2.5 Mid-level Intermediate Representation 5
2.6 Places 8

3 Rustlantis **10**
3.1 Overview 10
3.2 Types 11
3.3 Statements and declarations 12
3.4 `PlaceTable` 13
3.5 Terminators and transparent control flow 16
3.6 Ensuring reducible control flow 19
3.7 Representing memory layout 20
3.8 Pointer offsets 22
3.9 Picking "interesting" places 23
3.10 Producing observable and deterministic output 24

4 Evaluation **28**
4.1 Testing backends 28
4.2 Bugs discovered 29
4.3 Compiler code coverage 33
4.4 Performance 37
4.5 Energy efficiency 39

https://t.me/learningnets

5 Future Work **40**
5.1 Missing language constructs 40
5.2 Program reduction 41
5.3 Keep Rustlantis running 42

Bibliography **43**

A Proof of reducibility **46**

https://t.me/learningnets

Chapter 1

Motivation

Rust emphasises memory safety and logical correctness thanks to its borrow checker and strong type system. These promises rely on the Rust compiler correctly compiling the user's code. But sometimes a compiled program has the incorrect behaviour due to a bug in the compiler. These bugs are called *miscompilations*.

A long-standing miscompilation in the Rust compiler was the incorrect removal of loops even if the compiler cannot tell whether it terminates (#28728). This resulted in the Rust compiler apparently "proving" the unresolved Collatz conjecture by optimising this function to return `true` [1].

```
pub fn collatz(n: usize) -> bool {
  match n {
    1                => true,
    n if n % 2 == 0 => collatz(n/2),
    _                => collatz(3*n + 1),
   }
}
```

The source of this bug is LLVM [2] which Rust relies on for optimisations and machine code generation. It was originally developed for C and C++, but was later adopted by compilers of other languages such as Rust and Swift – these are called *language frontends*. The C/C++ compiler Clang is by far the most mature frontend. Other frontends, such as Rust, often enter code paths in LLVM not used by Clang and therefore ill-exercised and potentially bug-ridden.

Infinite loops and recursions are Undefined Behaviour in C and C++, therefore the compiler can assume that a loop always terminates. This does not hold for Rust. While LLVM does not require loops to terminate and should compile Rust loops correctly, its code contained assumptions that only hold for C/C++, thus miscompiling Rust code.

https://t.me/learningnets

Another infamous example is the `noalias` attribute on function parameters which allows LLVM to produce more efficient code, but this is rarely used when compiling C/C++. Multiple LLVM bugs have been causing the miscompilation of this attribute, with one fixed and a new one discovered, leading Rust to flip-flop between emitting and omitting this attribute:

- Feb 2016: Workaround LLVM optimizer bug by not marking `&mut` pointers as `noalias` #31545
- May 2018: Emit `noalias` on `&mut` parameters by default #50744
- Sep 2018: Do not put `noalias` annotations by default #54639
- Mar 2021: Enable mutable `noalias` for LLVM ≥ 12 #82834

For most software, the severity of a bug may be high or low, but its impact surface is generally predictable. However, compilers are so fundamental that the effect of a compiler bug is hard to predict. At best, it could manifest as a compiler crash or trigger a spurious failure in a test suite. At worst, a compiled program could silently produce wrong results in production, and the consequences are unbounded.

We would like to be able to proactively find bugs in the Rust compiler and fix them before they hit any real users. The Rust compiler has an extensive test suite, but the above examples have shown that this alone is not sufficient. Here, we will apply another testing technique to the Rust compiler: fuzzing.

https://t.me/learningnets

Chapter 2

Background

2.1 Fuzzing

Recognising the importance of compiler correctness, a range of techniques for automated compiler testing have been developed over the years. Fuzzing has been found to be a successful approach, finding hundreds of bugs in widely-relied compilers such as GCC and Clang [3, 4, 5].

The idea of fuzzing is fairly simple: we produce a test program, compile it, and observe the behaviour of the compiled program. If it is incorrect, then there is a bug in the compiler.

2.2 Differential testing

But how could we tell whether the compiler performed correctly given a random program? This is known as the *test oracle problem* [6].

We know that a compiler should never crash by e.g. a segfault or a panic. If the program contains an error, it should print a diagnostic message and exit gracefully. It is trivial to observe if the compiler crashed.

However, if the compilation succeeds, we do not know the correct runtime behaviour of a random program, and cannot easily say whether a miscompilation occurred.

To address this, we use a technique called *differential testing* [7], where we compile and run (or interpret) the same random program by different compilers or interpreters and under different optimisation settings – we call them *testing backends*. Given a deterministic and UB-free program, all execution results should be identical. If they differ, then at least one testing backend has a bug, which can then be manually identified.

https://t.me/learningnets

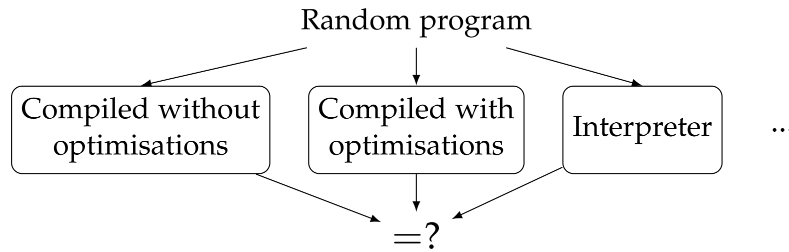


Figure 2.1: Visualisation of differential testing

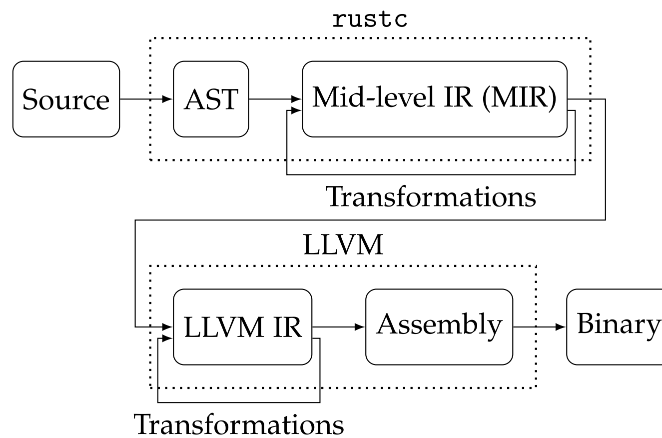


Figure 2.2: Rust's compilation pipeline

Determinism and UB-freedom are crucial to differential testing. Otherwise, a difference in observed output is not an indication of a bug in any testing backend.

2.3 The Rust compiler

The Rust compiler, `rustc`, compiles a Rust program in several stages, as represented by Figure 2.2: it first parses the program into an Abstract Syntax Tree (AST). Then, after several rounds of lowering, produces the Mid-level Intermediate Representation (MIR). It performs some analyses and optimisations on MIR, then lowers this into another intermediate representation for a *codegen backend*, which is responsible for further optimisations and the generation of machine instructions. The default and most commonly used codegen backend is LLVM, but alternatives are also available (such as Cranelift).

A lot of things happen in this pipeline, especially when optimisations are enabled. To compile a Hello World program at the "release" optimisation level (`-C opt-level=3`), `rustc` runs 80 passes on MIR and LLVM runs a

https://t.me/learningnets

whooping 1235 passes. If a bug is triggered in any one of these passes, then it could result in a miscompilation. We are therefore most interested in testing these optimisation stages.

2.4 Rust fuzzing

RustSmith [8] is a random Rust program generator. It is capable of generating programs utilising a wide range of Rust language constructs, including assignments, arithmetic operations, references, functions, loops, and many more. However, it only produces *safe* Rust programs. This automatically guaranteed that the program is UB-free as Rust guarantees that all safe programs are UB-free. But it also prevented RustSmith from producing `unsafe` operations, such as raw pointer dereference and transmutation (bit casting). These could exercise more edge cases in the compiler, as most programs written in Rust do not use `unsafe`.

RustSmith was able to trigger bugs in then-current and historical versions of Rust, but it did not discover any previously-unknown ones. This made it of limited use in proactively finding and fixing bugs.

We could extend RustSmith to produce `unsafe` operations. However, Rust is not a small language and `unsafe` especially imposes some very intricate constraints to a program it must uphold to avoid UB.

We propose a new fuzzer named **Rustlantis** with a new approach: instead of generating normal (surface) Rust programs, we target the Mid-level IR instead.

MIR encapsulates the semantics of a Rust program using a small set of constructs and contains little implicitness and syntax sugars found in surface Rust. This makes UB much easier to reason about. Nonetheless, compiling a MIR program still exercises the most complex and error-prone parts of the pipeline, namely optimisation passes performed by `rustc` and LLVM.

Therefore, MIR is an ideal fuzzing target for Rustlantis.

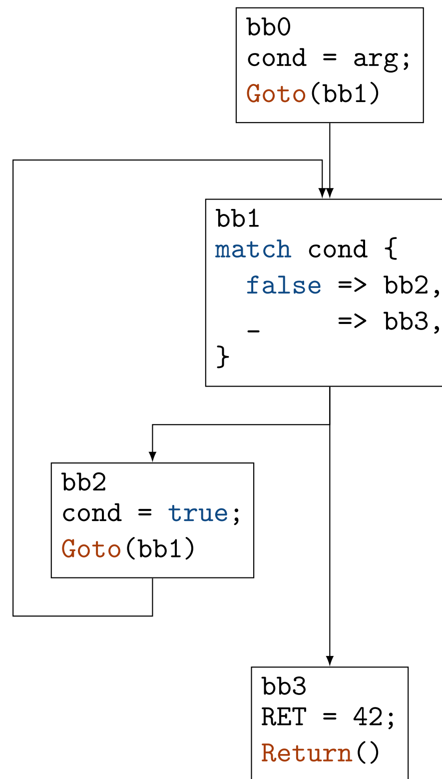
2.5 Mid-level Intermediate Representation

MIR is a control-flow graph (CFG) where each function body contains one or more successor *basic blocks*, in turn containing zero or more statements. Statements within a basic block are to be executed top-down with no branching or function calls. A basic block always ends with one *terminator* which can lead to one or more 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. Figure 2.3 shows an example function in MIR.

https://t.me/learningnets

```
fn f(arg: bool) -> i32 { mir!(
  let cond: bool;
  {
    cond = arg;
    Goto(bb1)
  }
  bb1 = {
    match cond {
      false => bb2,
      _     => bb3,
    }
  }
  bb2 = {
    cond = true;
    Goto(bb1)
  }
  bb3 = {
    RET = 42;
    Return()
  }
)}
```

(a) As code



(b) As control flow graph

Figure 2.3: An example MIR function

There are two textual formats for MIR: the output format of `--emit=mir` flag of `rustc`, and the custom MIR format [9]. The main difference is that the `--emit=mir` format is output-only and cannot be compiled by `rustc`, whereas custom MIR can be. Custom MIR can also be mixed with functions written in surface Rust syntax in the same file, and can be configured to be injected into different stages of the compilation pipeline. For these reasons, we will be generating the custom MIR format.

The grammar of the subset of MIR we will be generating is as follows:

```
pub enum Statement {
    /// a = b;
    Assign(Place, Rvalue),
}
pub enum Rvalue {
    /// x
    Use(Operand),
    /// !x
    UnaryOp(UnOp, Operand),
```


https://t.me/learningnets

```
    /// x + y
    BinaryOp(BinOp, Operand, Operand),
    /// x as i32
    Cast(Operand, Ty),
    /// Checked(x + y)
    CheckedBinaryOp(BinOp, Operand, Operand),
    /// addr_of!(x) or addr_of_mut!(x)
    AddressOf(Mutability, Place),
    /// (x, y) or [x, y, z] or Foo { a: x }
    Aggregate(AggregateKind, IndexVec<FieldIdx, Operand>),
}
pub enum Operand {
    Copy(Place),
    Move(Place),
    Constant(Literal),
}
pub enum UnOp { Not, Neg }
pub enum BinOp {
    Add, Sub, Mul, Div, Rem,
    BitXor, BitAnd, BitOr,
    Shl, Shr,
    Eq, Lt, Le, Ne, Ge, Gt,
}
pub enum Mutability { Not, Mut }
pub enum AggregateKind {
    Tuple,
    Array(Ty),
    Adt(Ty),
}
```

The `Place` component appears very often in MIR. It represents a memory location. We will talk about this in detail in Section 2.6.

The main differences between custom MIR and surface Rust are:

- The RHS of an assignment (an `Rvalue`) can perform only one "operation". For instance, chaining multiple arithmetic operations is not allowed.
- Everything is implicitly `unsafe`. Unsafe operations in surface Rust, such as dereferencing raw pointers, can be carried out directly.
- All local variables are declared at the top of the function[1] with a type annotation. They are uninitialised until written to.

[1] This is not strictly required, but we do this for consistency

https://t.me/learningnets

- Function parameters are functionally the same as local variables declared with let, except that they are initialised with the value supplied by the caller upon function entry.
- All local variables and parameters are declared as-if with a mut binding and can be assigned multiple times[2].
- There is a special local variable RET which refers to the return slot. It can be read from or written to anywhere in the function, and its value is copied to the caller upon return. In surface Rust, there is no way to access the return slot directly.
- Arithmetic and bitwise operations are wrapping and cannot panic. Checked can be used to perform checked addition, subtraction, and multiplication. It produces a tuple ({integer}, bool), where the boolean indicates the check result.
- There is no Deref coercion. Dereferencing must be explicit.

MIR has more than a dozen types of terminators. We will focus on these four:

- Goto(BasicBlock): Unconditionally enter another basic block
- SwitchInt(discriminant: Place, switch_targets: [(<literal>, BasicBlock)], fallthrough: BasicBlock) or match: Evaluate the value of the discriminant, then enter the basic block with the matching value. If nothing matches, enter the fallthrough basic block.
- Call(return_place: Place, target: BasicBlock, function: Function, arguments: [Operands]): Call another function, then enter target after it has returned.
- Return(): Copy the value of the return slot to the return_place in the corresponding Call terminator, then return from the current function and enter the target basic block.

2.6 Places

The concept of *place expression* is important in MIR. This is similar to lvalues in C. Syntactically, a place expression contains a local variable and zero or more *projections*. Rustlantis can generate 3 types of projections:

- Tuple or struct field: a.0, a.foo
- Primitive array index: a[0], a[i]
- Pointer dereference: *a

[2] This means that MIR is not in single-static assignment form

https://t.me/learningnets

Arbitrarily many (or none) projections can be chained together in a place expression. There is no length restriction. However, one caveat of the MIR dialect Rustlantis will generate is that if a dereference is part of the place expression, then it must be the first projection. For instance, `*(a.foo)` is not allowed, as the dereference is after a field projection.

A *place* refers to the memory location a *place expression* evaluates to at a point in the program. Due to pointers, syntactically identical place expressions may evaluate to different places at different points in the program. For instance,

```
1 let x: (i32, i32);
2 let y: (i32, i32);
3 let a: *mut (i32, i32);
4 {
5     a = addr_of_mut!(x);
6     (*a).0 = 42;
7     a = addr_of_mut!(y);
8     (*a).0 = 42;
9 }
```

`(*a).0` on line 6 is syntactically identical to `(*a).0` on line 8, but the former refers to the same location as `x.0`, whereas the latter refers to `y.0`. We also note that pointers may alias, so there may be multiple place expressions referring to the same location.

https://t.me/learningnets

Chapter 3

Rustlantis

> The name *Rustlantis* is a reference to Space Shuttle *Atlantis*, which docked with space station *Mir* 7 times under the Shuttle–*Mir* program.

3.1 Overview

Rustlantis can generate programs containing the following MIR constructs:

- Types: All primitive integer and floating point types, `bool`, `char`, arrays, tuples, raw pointers, and structs.
- Functions.
- Terminators: `Goto`, `Return`, `SwitchInt` (`match`), `Call`.
- Intrinsic functions: `arith_offset` (for pointer arithmetics) and `transmute` (for bit-casting).
- Operators: all arithmetic, logical and bitwise operations on integers and floating points, and checked arithmetic operations on integers.
- All primitive literal expressions, as well as tuple, array, and struct aggregate expressions.
- Creating pointers with `addr_of!` and `addr_of_mut!`, and dereferencing them.
- Casts between integers, floating points, `char`, and `bool`.

Rustlantis takes a seed as the command-line argument. It is guaranteed to always produce the same program with a given seed.

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 [10], a Rust interpreter.

https://t.me/learningnets

By design, programs generated by Rustlantis are terminating, deterministic, and free from Undefined Behaviour under the Tree Borrows [11] aliasing model. 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.

3.2 Types

Composite types, such as tuples, arrays, and structs can contain other types which can in turn also be composite. This meant that the amount of possible types in a Rust program is infinite.

Rustlantis determines a finite amount of types by first populating the *typing context*, which contains all the types that may appear in the subsequently generated program. The typing context is first populated with primitive types, and then raw pointers, tuples, and structs are iteratively added by randomly combining existing types. These compound types can nest arbitrarily up to a depth limit, but structs cannot transitively contain a field of its own type.

By generating all the types in advance, instead of on the fly, Rustlantis can assign each type a fixed weight, controlling the distribution of variable types in the program. This allows us to increase the frequency of more "interesting" types. For instance, the weights of types that transitively contain a pointer sum up to 20%, in contrast to integer primitives which sum up to 10%. `isize` alone weighs 10%, as this is the type used for pointer offsets and index projections. Table 3.1 shows a selection of types generated and their assigned weights.

Type	Weight
`isize`	10%
`i8`	1.00%
`[i8; 8]`	1.00%
`*const u8`	1.37%
`(i16, i8, *const u8)`	1.37%
`*mut (i16, i8, *const u8)`	1.37%
`struct Adt50 {fld0: u64, fld1: char}`	1.00%
`struct Adt61 {fld0: Adt50}`	1.00%
...	

Table 3.1: A partial selection of types in a seeded typing context and their weights

https://t.me/learningnets

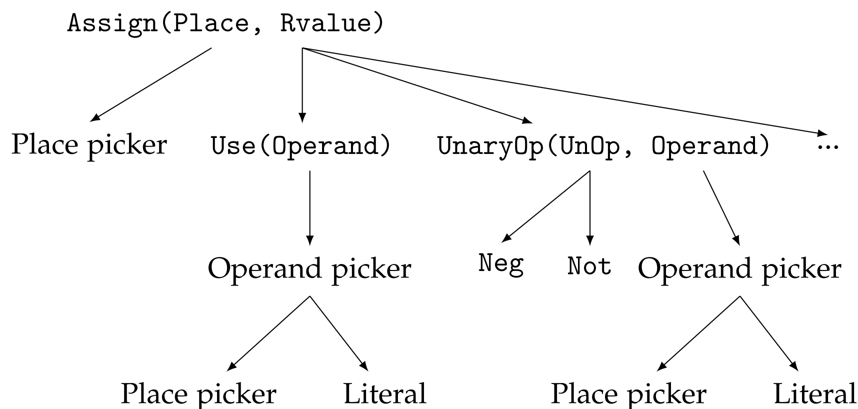
3.3 Statements and declarations

Rustlantis bootstraps by generating the entry function, `fn0`, with a list of parameters of random types. `fn0` is called in `main` supplied with random literals. Each literal is wrapped in `std::hint::black_box()`, so the compiler cannot optimise using their values.

The generation process of Rustlantis is similar to an interpreter or CPU, but instead of executing code, it is producing them. 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 `fn`x`:bb`y). All new statements are added to the end of a basic block and existing statements are never modified, therefore Rustlantis does not need to keep track of the statement index in a basic block.

Rustlantis first enters `fn0:bb0`. At each step, Rustlantis decides whether it should generate a statement or declaration.

Each statement is generated by recursively descending on its syntactical components:



For an assignment statement, Rustlantis needs to choose two sub-components: a `Place` and an `Rvalue`. The `Place` is the left-hand side of the assignment. A dedicated function will provide a list of candidate places, which is described later in Section 3.4.

Assuming for now that we have chosen a `Place` from the candidate list, we call a function to generate the `Rvalue`. This is not always successful. For instance, if the LHS `Place` is a pointer, but there is no expressible place in the current function with the pointee type, then it is impossible to assign any `Rvalue` to the LHS. When this happens, Rustlantis removes the chosen `Place` from the candidate list, picks another place and then tries again. This repeats

https://t.me/learningnets

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

This choose-remove-retry process can be expressed as the following pseudo-code function

```
function generate_assignment(lhs_candidates: [Place]):
  while lhs_candidates.not_empty()
    lhs = lhs_candidates.pick()
    case generate_rvalue(lhs)
      rvalue => return Statement::Assign(lhs, rvalue),
      error  => lhs_candidates.remove(lhs),
    end case
  end while
  // We have no candidate left
  return error
end function
```

The function which generates an `Rvalue` has 7 variant candidates to choose from. The choice is made in the same choose-remove-retry manner as the `Place` choice: a random variant is chosen and a function is called to try to generate an `Rvalue` of this variant, if this fails, then the variant is removed from the candidate list and the process starts again.

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 expression.

After recursively trying all options, the root which generates the `Assign` statement may run out of place candidates. In this case, a new local variable is declared in the current function, and the type is randomly chosen using the weights in the typing context.

3.4 `PlaceTable`

Recalling the definition of a *place expression*: a local variable chained with a series of projections. This naturally gives rise to a graph structure, where nodes are places, and edges are projections from one place to another.

The data structure `PlaceTable` is responsible for keeping track of all places in the program, globally. Figure 3.1 shows a representation of the `PlaceTable`'s content when the cursor is on line 7.

When a local declaration is generated, the local is added to `PlaceTable`, along with all the places reachable from the local through projections. `Deref` projection edges are added not on allocation of a pointer, but when a statement assigning to the pointer is generated. We know at all times where an

https://t.me/learningnets

initialised pointer is pointing to. If a pointer is overwritten, the old `Deref` edge is removed and a new one is added targeting the new pointee.

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

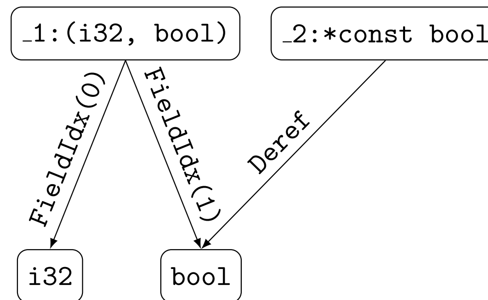


Figure 3.1: `PlaceTable` contents

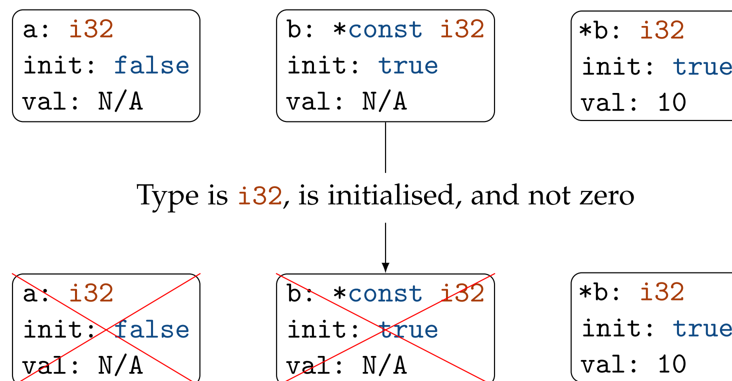
Note this isn't quite the right memory model for Rust: unlike C and C++, Rust does not have strict aliasing or "typed memory". This means that a local variable does not contain a fixed set of places upon creation, and the same memory location can be referred to by different place expressions of different types (assuming the alignment constraint is satisfied). Nonetheless, `PlaceTable` is a valid, albeit restricted, approximation to Rust's full memory model.

Each node in `PlaceTable` also keeps track of some information about the runtime state of the place, such as whether it is initialised. In case a literal assignment to it is generated, the value of the literal is saved as the *known value* of the place (until it is overwritten by a non-literal). The information in a node can be propagated to another node when an operation that copies between places is generated.

Whenever the program generator needs a place, we gather all local variables and perform a depth-first search through `PlaceTable` starting from each of them. This gives us the full list of expressible places. Then, using the information in `PlaceTable`, 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 with the same type as the dividend, is initialised, and has a non-zero known literal value.

https://t.me/learningnets



The filtering guarantees that all place candidates are valid 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 add a stack data structure to `PlaceTable` alongside the existing graph. The stack contains a list of `Frame`s, which represents a call frame.

Each `Frame` contains a list of nodes in the `PlaceTable` 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.

Besides locals, each `Frame` additionally contains the return place node in the caller[1], and a list of arguments that were `Move`d into the current function in the `Call` terminator.

We need to know the node for the return place in the caller because it is UB to access it until the function returns. Similarly, it is also UB to access a place that is a `Move` argument until the function returns, so we keep track of these in the `Frame`.

When we are picking a place, we filter out return place and `Move`d-in nodes in all `Frame`s in the call stack, as illustrated by Figure 3.2.

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 `PlaceTable`. Deallocated places can never be chosen. This prevents UB if a function returns a pointer to a local variable.

[1]This must be a specific *node* in `PlaceTable`, not a place expression. This is because the return place expression may evaluate to different places before and after the call if it contains a dereference. In MIR, the return place expression is evaluated before entering the call.

https://t.me/learningnets

```
1  fn fn1() { mir!(
2    let _1: i32;
3    let _2: *mut i32;
4    let _3: Foo;
5    let _4: *mut Foo;
6    {
7      _2 = addr_of_mut!(_1);
8      _3 = Foo { /* ... */ };
9      _4 = addr_of_mut!(_3);
10     Call(_1, bb1, fn2(_2, Move(_3), _4))
11   }
12   bb1 = {
13     // Return target
14   }
15 )}
16
17 fn fn2(
18     _1: *mut i32,
19     _2: Foo,
20     _3: *mut Foo
21 ) -> i32 { mir!(
22   {
23     // <- Current Cursor
24   }
25 )}
```

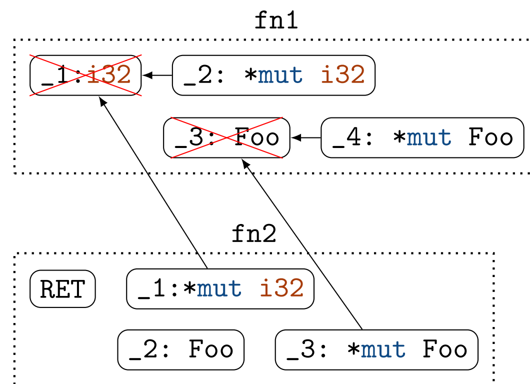


Figure 3.2: `PlaceTable` contents with multiple functions

3.5 Terminators and transparent control flow

Once a random amount of statements is reached in a basic block, a terminator is generated, and the generation of the basic block is complete. Rustlantis needs to resume its generation by setting the `Cursor` elsewhere. To keep the information in `PlaceTable` accurate at all times, Rustlantis needs to generate the program in the exact same order as it would be executed. This means that we need to resume generation in the same basic block as the one the new terminator would take us. We will talk about how we guarantee this for each of the 4 terminators.

`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.

https://t.me/learningnets

SwitchInt

SwitchInt is the most complex terminator, as there are multiple potential successors. Our strategy is illustrated by Figure 3.3 and described below.

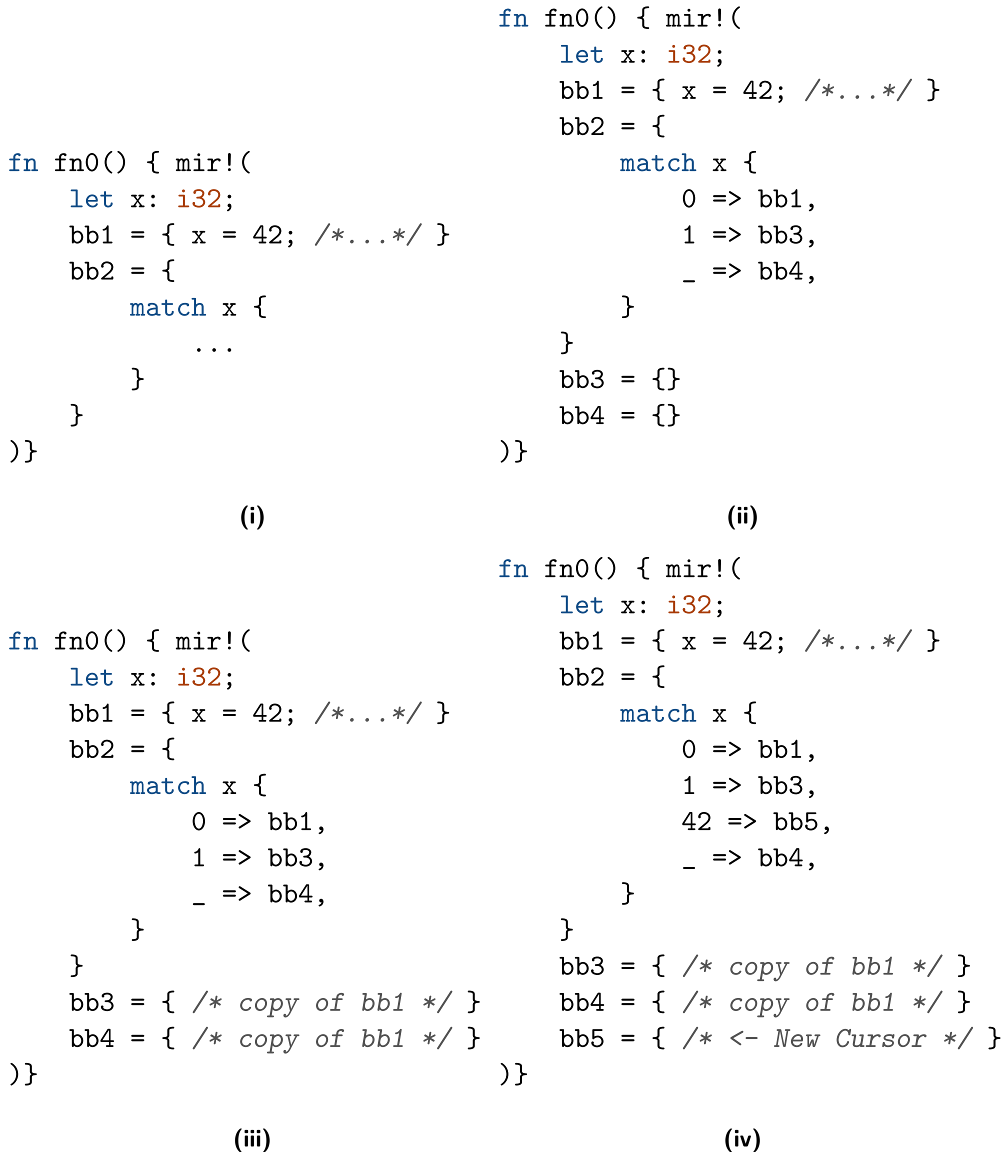


Figure 3.3: Steps of generating a SwitchInt

(i) First, we pick a place with a known value as the discriminant.

(ii) Then, we generate a list of *decoy* arms paired with literals that we know are not the known value of the discriminant. The target of a decoy arm can be a random existing basic block like `bb1`, or new empty basic

https://t.me/learningnets

blocks like `bb3` and `bb4`.

(iii) If we have added new decoy basic blocks, we fill them with an identical copy of the content in 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 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 and pair it with the known value of the discriminant. 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 `PlaceTable`.

Although Rustlantis knows the precise value of the discriminant, the compiler cannot always spot, and sometimes must not exploit this information. For instance, the discriminant 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. And a call could be repeated in decoy basic blocks, 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.

This strategy is similar in spirit to the Equivalence Modulo Inputs [12] approach to compiler testing. EMI mutates code paths which are not executed under a given input, thus preserving the program semantics under that input. But a compiler cannot statically know which code path will be taken, allowing the mutated parts to trigger miscompilations that change executed the program semantics. Our decoy arms and basic blocks serve a similar purpose to the mutated code paths in the EMI approach.

`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

https://t.me/learningnets

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.

We also push a new `Frame` to `PlaceTable`'s call stack containing the return place node and all `Move` arguments generated in the terminator.

`Return`

When a `Return` terminator is selected, we must first check `PlaceTable` to see if the return slot `RET` is initialised, as it is UB to return from a function with an uninitialised return slot, even if the return value is never used in the caller. If `RET` uninitialised, we cannot produce a `Return` terminator, so we return an error to the caller to select another terminator instead.

Once the function returns, we

1. Copy `RET`'s `PlaceTable` node content to the return place.
2. Pop the last `Frame` off the `PlaceTable` call stack.
3. Mark all places reachable with non-`Deref` projections from locals in the poped `Frame` as deallocated in `PlaceTable`.
4. Set our cursor to the return target basic block in the caller and resume generation.

3.6 Ensuring reducible control flow

A *reducible* control flow graph is a CFG that can be expressed using loop, if-else, and break statements only. Irreducible control flow is only possible in the presence of goto statements. Since surface Rust does not have goto statements, all MIR generated from surface Rust programs are reducible. However, it is possible to write custom MIR programs with an irreducible CFG. The canonical example of an irreducible CFG is a loop with two entries, as shown in Figure 3.4

Currently, there is no formal requirement that MIR must be reducible at all times. Some optimisations can in fact make MIR irreducible[2].

Nonetheless, MIR from surface Rust before any optimisations will remain reducible for the foreseeable future as there is no plan to add goto into the language, and some passes may rely on this fact.

[2] https://github.com/rust-lang/rust/issues/114047#issuecomment-1649793275

https://t.me/learningnets

```
fn f(arg: bool){ mir!(
  {
    match arg {
      true => bb1,
      _    => bb2,
    }
  }
  bb1 = {
    Goto(bb2)
  }
  bb2 = {
    Goto(bb1)
  }
)}
```

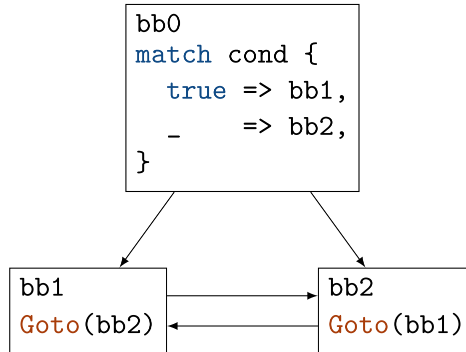


Figure 3.4: An irreducible CFG in MIR

Rustlantis guarantees the generated program has a reducible control flow. We provide a formal proof in Appendix A.

3.7 Representing memory layout

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 (other than requiring elements to be aligned). Figure 3.5 shows some possible memory layouts of `(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.

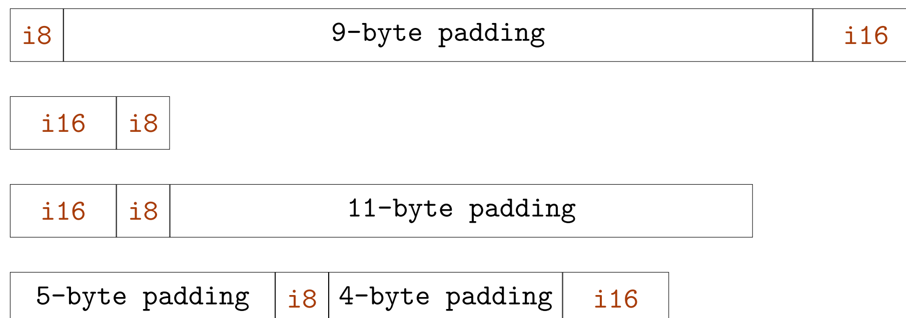


Figure 3.5: Some possible memory layouts of `(i8, i16)`

Reading padding bytes is UB. We usually don't need to worry about where

https://t.me/learningnets

the padding bytes are as `PlaceTable` does not contain any place expression overlapping with paddings.

However, the `transmute` intrinsic requires the source and destination to have the same size (otherwise it's a compile error). Additionally, it reads the from source place according to the destination's type layout, and it must not read padding bytes from source (otherwise it's UB).

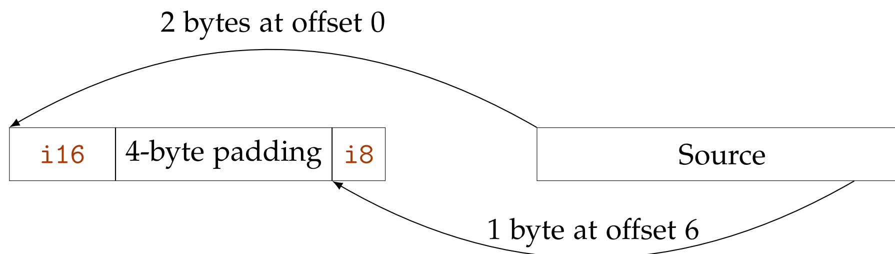


Figure 3.6: Representation of copying by `transmute`

We can `transmute` a fully initialised value to a destination of the same type, as they are guaranteed to have the same size and padding (which are never accessed). But this is trivially optimisable to a normal copy assignment by the compiler.

To support transmutations between places of different types, we need to represent the size and memory layout of places in our `PlaceTable`. 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 to `Sized` types (which are all the types in Rustlantis), 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. Places of types fitting into a `Run` are associated with one in its `PlaceTable` node. In the case of arrays, bytes in its `Run` are shared with its elements since both the array type and its element types have known sizes and are paddingless. Figure 3.7 shows a representation of `PlaceTable` with `Run`s.

When we are picking the argument to a `Call` terminator calling `transmute`, we restrict the place candidates to ones that have an associated Run of the same size.

`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` can contain invalid bit

https://t.me/learningnets

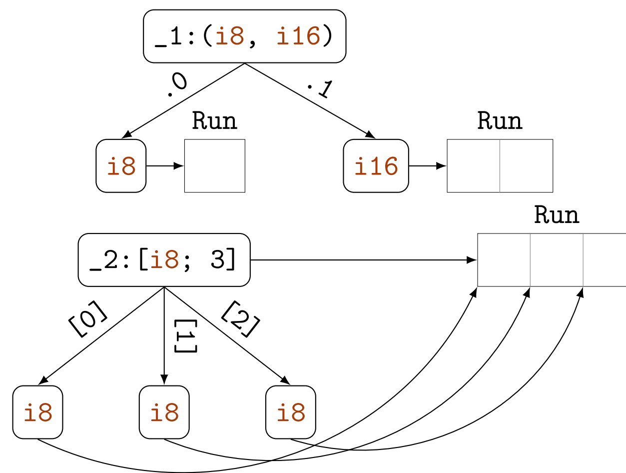


Figure 3.7: Places pointing to `Runs`

patterns. 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.

3.8 Pointer offsets

We use `arith_offset` to offset pointers, 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). We only need to guarantee the validity of the offsetted pointer when it is later dereferenced.

We only pick places with known values as the offset argument but without restrictions on its value. We keep track of the offset amount from its original pointee in the `PlaceTable` node of each pointer-typed place, accumulating with the known value offset on each call to `arith_offset`.

Whenever we are walking `PlaceTable` to find place candidates, we only visit projection (`Deref`) of pointers whose offset value is 0. The offset value is zero if the pointer has never been offsetted, 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. Unfortunately, we did not have time to implement this.

https://t.me/learningnets

3.9 Picking "interesting" places

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

Additionally, we assume that complex series of operations are more likely to trigger miscompilations. Therefore, we would like to expose the results of complex operations to the output values. Printing out 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 field `complexity` is added to each node in `PlaceTable`, 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
- `Move` and `Copy` operands take the complexity value from its underlying `Place`
- `Rvalue::AddressOf` 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 favour 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 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 favour places containing a dereference projection, and especially favour places dereferencing a pointer that has been "roundtripped" by pointer offsetting.

We perform additional weight augmentations depending on the selection context: if the place is for a function argument, we favour places that are

- pointers,

https://t.me/learningnets

- offsetted pointers, or
- places with known values.

If the place is for the LHS of an assignment, we ignore its complexity, but favour places that are uninitialised.

Besides place picking, we also weigh the selection of all other syntactical choices, including `Rvalues`, `BinOps`, `UnOps`, and so on. For instance, we do not want to generate `Use` too often because it is not very interesting, but we do want to generate `AddressOf` often as pointers are complex to optimise. But unlike weights of places, these weights are empirically determined, hard-coded, and not influenced by the state of the generated program.

3.10 Producing observable and deterministic 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. However, the states in programs are very often non-deterministic. For instance, allocating a stack variable is regarded as a non-deterministic action as the address depends on the OS and is (often intentionally) unpredictable. Should non-determinism be exposed through the output, differential testing will encounter a large number of false negatives. So we must ensure only deterministic values can influence the output.

We categorise types into two groups: ones whose values (bit patterns) are always deterministic, and ones which may be non-deterministic. Of the types that may be produced by Rustlantis, two kinds may have non-deterministic values.

1. Pointers: their values are determined by the OS at runtime.
2. Floating points: if a floating point operation produces a `NaN` value, its "payload" bits are unspecified.

Deterministic types are therefore all the types that are not and do not contain pointers and floating points.

Data can flow from deterministic types into non-deterministic ones without restriction, but dataflow from non-deterministic types into deterministic ones needs to be carefully controlled to prevent leakage of non-determinism into the observable outputs. There are three ways this leakage can happen:

- pointer-to-int casts and transmutes,
- pointer comparisons, and
- floating point-to-int transmutes (casts are deterministic as the `NaN` payload do not affect cast results).

https://t.me/learningnets

Rustlantis does not generate these three operations to guarantee that all values of the deterministic types can be part of the observable output.

It happens to be the case that all of the deterministic types implement (or can derive) the `Debug` trait. Before we generate a `Return` terminator, we choose some local variables weighed by their complexity values and call a gadget function, `dump_var` (Figure 3.8), to print them to the standard output.

```
#[inline(never)]
fn dump_var(
  f: usize,
  var0: usize, val0: impl Debug,
  var1: usize, val1: impl Debug,
  var2: usize, val2: impl Debug,
  var3: usize, val3: impl Debug,
) {
  println!("fn{f}:_{var0} = {val0:?}\n\
                 _{var1} = {val1:?}\n\
                 _{var2} = {val2:?}\n\
                 _{var3} = {val3:?}");
}
```

Figure 3.8: Slow `dump_var`

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. It is about 10% faster than dumping variables individually.

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 the differential testing speed. It made each generated program 2.6 times slower to run compared to ones where chosen variables are supplied to `std::hint::black_box` (so they are not optimised out) but not printed. The culprit is the interpreter Miri, which is very slow when printing a large amount of data to standard output. Instead of printing individual variables, we need to find a data structure that captures the value of each variable efficiently and results in a difference when any of the values are different.

The solution is hashing. It happens to be the case that all deterministic types also implement the `Hash` trait. We can modify our `dump_var` function to hash each value with a global hasher, as shown in 3.9. The hash value is printed

https://t.me/learningnets

only once before the program exits.

```
static mut H: DefaultHasher = DefaultHasher::new();

#[inline(never)]
fn dump_var(
  val0: impl Hash,
  val1: impl Hash,
  val2: impl Hash,
  val3: impl Hash,
) {
  unsafe {
    val0.hash(&mut H);
    val1.hash(&mut H);
    val2.hash(&mut H);
    val3.hash(&mut H);
  }
}

pub fn main() {
  fn0(/* fn0 arguments */);
  println!("hash: {}", unsafe { H.finish() });
}
```

Figure 3.9: Fast `dump_var`

The hash version only made the generated program 0.6% slower to run compared to the no-output baseline. However, this significantly reduces the debuggability of the generated program when it causes a miscompilation, as we do not have visibility over which variable in which function had different values. This made reducing the usually thousand-line-long program down to a minimal reproducible example difficult.

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.

As compiler optimisations are highly volatile, sometimes a small change can trigger or suppress a miscompilation. We minimise the chance of the bug being sensitive to the variant of `dump_var` by annotating it with the

https://t.me/learningnets

`#[no_inline]` attribute so that optimisations are unlikely across calls to `dump_var`. Nonetheless, there may still be programs that only result in a difference with the fast `dump_var`, but the bug disappears when it is tested again with the debug `dump_var`. In this case, we still have a reproduction and are still able to investigate the miscompilation, only more difficult. Some programs may only trigger observable miscompilations with the debug `dump_var`. We will miss these. However, fuzzing through programs far more quickly likely results in overall more bugs being discovered, so this is a worthy trade-off to make.

https://t.me/learningnets

Chapter 4

Evaluation

4.1 Testing backends

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

1. Compiled with MIR optimisations and LLVM optimisations
2. Compiled with only LLVM optimisations
3. Compiled with only Cranelift optimisations
4. Interpreted with Miri

Cranelift [13] is a machine code generator developed by Bytecode Alliance. It translates Cranelift IR into target-specific machine instructions. rustc can generate Cranelift IR and use Cranelift as the machine code generator as an alternative to LLVM IR and LLVM. It is comparatively new to LLVM and far less widely used, therefore we hypothesised that it may contain more bugs due to its immaturity.

Miri [10] is a Rust interpreter which executes MIR. It can detect and report UB encountered during 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 over tens of millions of programs on the most recent versions of Rustlantis and no UB has been reported.

Miri interprets MIR using the compile-time function evaluation (CTFE) mechanism implemented in `rustc`. CTFE is also used for the optimisation of compiled code by evaluating known values at compile time. So a bug in Miri can indicate a potential miscompilation too.

The combination of testing backends allows us to detect bugs in rustc, LLVM, Cranelift, and Miri/CTFE: A bug in rustc can cause backend 1 to be different.

https://t.me/learningnets

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/CTFE can cause backends 1 and 4 to be different.

4.2 Bugs discovered

Fuzzing with Rustlantis is trivially parallelisable as multiple instances can be running simultaneously with different seeds using GNU Parallel [14]. It is therefore well-suited on HPC clusters. Rustlantis has been fuzzing on the Euler cluster of ETH Zürich, which is an x86_64-based Linux platform. **13 previously-unknown bugs** have been discovered after 4.5 CPU years of fuzzing. Table 4.1 provides a breakdown of these bugs by the origin of the bug. Table 4.2 lists all bug reports filed to the relevant projects, including two reports whose root causes have been previously reported.

	Miscompilation	Crash
rustc	3	2
LLVM	6	2
Cranelift	0	0

Table 4.1: Overview of previously-unknown bugs discovered by Rustlantis

9 out of the 13 previously-unknown bugs are miscompilations, the most serious type of compiler bug. LLVM contained the most bugs, this is unsurprising as it is the most complicated part of the compilation pipeline by far.

Despite being new, Cranelift held up well with no bugs discovered by Rustlantis. Of course, this is not proof that Cranelift is overall more correct than LLVM, but it does serve as evidence in favour of Cranelift. This is likely because it performs little optimisations and its developers have dedicated fuzzing and formal verification efforts to ensure its correctness [13]:

> [Cranelift] is carefully fuzzed as part of Wasmtime with differential comparison against V8 and the executable Wasm spec, and the register allocator is separately fuzzed with symbolic verification. There is an active effort to formally verify Cranelift's instruction-selection backends.

We'll take a closer look at a selection of the bugs discovered by Rustlantis in the following subsections.

https://t.me/learningnets

Date	Report	Type	Origin	First report
24 March	rust#109567	Miscompilation	rustc	rust#102403
Const eval gives $x\%x$ wrong sign when x is a negative float				
27 April	rust#110902	Crash	rustc	New bug
Assertion failure in `RenameReturnPlace`				
28 April	rust#110947	Miscompilation	rustc	New bug
`ConstProp` propagates over mutating borrows				
10 May	rust#111426	Crash	rustc	New bug
`ReferencePropagation` prevents partial initialisation				
12 May	rust#111502	Miscompilation	rustc	New bug
`*const T` in function parameters annotated with `readonly`				
29 May	rust#112061 llvm#63019	Miscompilation	LLVM	New bug
Aliasing analysis merges loads from different offsets				
30 May	llvm#63013	Crash	LLVM	New bug
Phi nodes assumed to be non-empty				
31 May	llvm#63033	Crash	LLVM	New bug
Assertion failure in `RegisterCoalescer`				
1 June	rust#112170 llvm#63055	Miscompilation	LLVM	New bug
Constant folding produces invalid boolean values				
2 June	rust#112213	Miscompilation	LLVM	llvm#51838
Write to dangling pointer is hoisted before branching				
11 June	rust#112526 llvm#63266	Miscompilation	LLVM	New bug
Aliasing analysis is broken for overflowing pointer offsets				
12 June	rust#112548	Miscompilation	LLVM	New bug
Safe program miscompiled on Apple silicon macOS				
18 June	rust#112767 llvm#63430	Miscompilation	LLVM	New bug
Copy elision corrupts stack arguments with two parts				
23 June	llvm#63475	Miscompilation	LLVM	New bug
Copy elision reads stack arguments from the wrong offsets				
6 July	rust#113407	Miscompilation	rustc	New bug
Subnormal `f64` to `f32` cast is wrong				

Table 4.2: All bugs encountered by Rustlantis

https://t.me/learningnets

Wrong comparison of `AliasResults`

The original generated program had over 6,500 lines. We reduced and rewrote it into surface Rust and filed rust#112061 on 29 May. A Rust contributor @Nilstrieb further reduced it into LLVM IR and filed llvm#63019.

This is the minimal reproducible example:

```
1  define i8 @test(i1 %c, i64 %offset, ptr %ptr) {
2  start:
3    %alloca = alloca [8 x i8], align 8
4    store i64 u0x1122334455667788, ptr %alloca, align 8
5    %gep.2 = getelementptr i8, ptr %alloca, i64 2
6    %gep.dynamic = getelementptr i8, ptr %alloca, i64 %offset
7    br i1 %c, label %join, label %if
8
9  if:
10   br label %join
11
12 join:
13   %phi = phi ptr [ %gep.dynamic, %start ], [ %gep.2, %if ]
14   store i8 0, ptr %alloca, align 8
15   %load1 = load i64, ptr %alloca, align 8
16   store i64 %load1, ptr %ptr, align 8
17   %load2 = load i8, ptr %phi, align 1
18   ret i8 %load2
19 }
```

Note that `%load1` on line 15 loads eight bytes from the stack allocation `%alloca` without any offset, and `%load2` loads one byte from the same stack allocation but is either offset by the constant 2 or the dynamic value `%offset`, depending on the taken branch. LLVM cannot say that `%load1` and `%load2` have the same initial byte, as the addresses could be different.

However, LLVM 16.0.4 incorrectly reasoned that `%load1` and `%load2` are loaded from the same address, turning `%load2` into a simple truncation of `%load1`

```
15 %load1 = load i64, ptr %alloca, align 8
16 store i64 %load1, ptr %ptr, align 8
17 %0 = lshr i64 %load1, 16
18 %1 = trunc i64 %0 to i8
19 ret i8 %1
```

The bug was serious: any load from the same allocation but different offsets may be incorrectly merged into one. The fix was merged into LLVM on 31 May and backported to LLVM 16.0.5 release.

https://t.me/learningnets

`undef` in booleans

The generated program has over 11,000 lines. We reduced this and got a pure-LLVM IR reproduction with the help of `llvm-reduce` [15]. We filed rust#112170 and llvm#63055.

An LLVM contributor @nikic further reduced this into a 3-line function

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

`0x7FF8000000000000` is a bit pattern of `NaN` under IEEE-754 double-precision floating-point format [16]. The instruction `fcmp une` checks if the operands are not equal. The value of `%fcmp` should be `true` as `NaN` does not equate anything (not even itself), therefore the function should return `1`.

However, LLVM 16.0.4 generates the following x86 assembly, which returns 255.

```
test:
    movl    $255, %eax
    retq
```

This is due to the `fcmp une` instruction being incorrectly folded into an 8-bit long `undef` constant, which can contain any bit pattern.

The fix was merged into LLVM on 1 June.

`ConstProp` across mutating pointer

Unlike the previous two, this is a miscompilation in `rustc`. The reduced MIR was rewritten into surface Rust and further reduced to this example:

```
1  pub fn fn0() -> bool {
2      let mut pair = (1, false);
3      let ptr = core::ptr::addr_of_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 optimisation incorrectly propagating the constant value of

https://t.me/learningnets

`pair.1` (`false`) from line 4 to line 8 in the presence of a live pointer to the place.

Notably, this reproduction has UB under the Stacked Borrows [17] aliasing model and therefore invalid, as `ptr` would have been invalidated by the assignment on line 4 and its later use on line 6 would not be allowed. But now that Tree Borrows has been implemented in Miri as an alternative to Stacked Borrows and allows this program, the Rust compiler must not miscompile it.

This also demonstrated the effectiveness of Rustlantis in exposing bugs that are hard to find in manual testing. One Rust project contributor said [18]:

> I've been suspecting such a bug without managing to reproduce it for a few months.

4.3 Compiler code coverage

Code coverage measures the number of statements and branches in a code base that have been exercised by a test suite. Coverage of the compiler source code is a metric commonly used by authors of existing compiler fuzzers [4, 5, 8] to measure how diverse the generated programs are.

We built **Rust 1.70.0** with coverage instrumentation enabled in both `rustc` (by using Mayank Sharma's patch which modifies `rustc`'s bootstrapping script [19]) and LLVM (with CMake flag `LLVM_BUILD_INSTRUMENTED_COVERAGE = "On"`). Using this instrumented compiler, we compiled 300 Rustlantis-generated programs with the maximum optimisation flags `-Z mir-opt-level=4 -C opt-level=3` to gather the coverage data. We then used `grcov`[1] and `lcov`[2] to process these coverage data.

We also generated 300 programs with RustSmith and gathered coverage information using the same Rust build and compilation flags to serve as a comparison with Rustlantis' numbers. The RustSmith thesis [8] contained some coverage statistics, but we will not use them for comparison as the Rust version and number of files tested are different.

We are most interested in the coverage of four parts of the compiler which we believe are the most bug-prone. These steps are labelled in red in Figure 4.1.

- `rustc`'s MIR passes
- `rustc`'s MIR lowering to LLVM IR
- LLVM's passes

[1] https://github.com/mozilla/grcov
[2] https://github.com/linux-test-project/lcov

https://t.me/learningnets

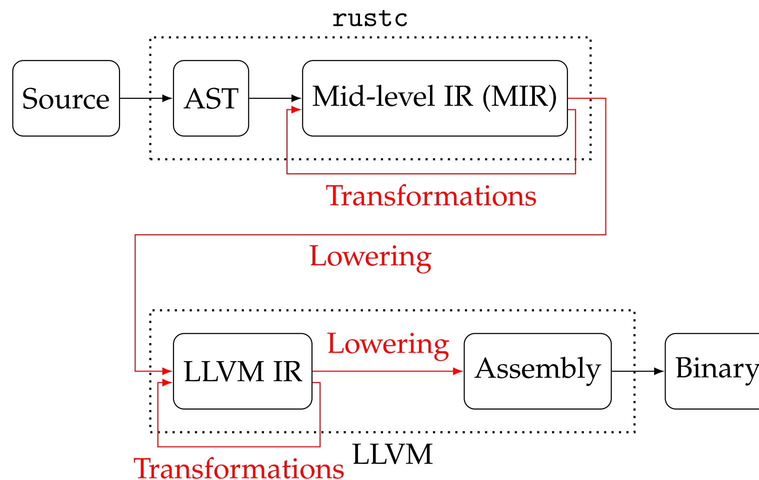


Figure 4.1: Coverage-measured parts

- LLVM's LLVM IR lowering to machine instructions

Unfortunately, Rust does not currently support branch coverage instrumentation[3], so we only have this information for code belonging to LLVM.

MIR optimisation

This crate lives under `compiler/rustc_mir_transform` directory of the Rust repository. It performs optimisations on MIR and has been responsible for 2 out of 3 `rustc` miscompilations and both `rustc` crashes discovered by Rustlantis.

		Rustlantis		RustSmith	
	Total	**Hit**	**Cov.**	**Hit**	**Cov.**
Lines	12638	5272	41.7%	5754	45.5%
Functions	1416	512	36.2%	563	39.8%

Table 4.3: Coverage of `rustc_mir_transform`

Lowering to LLVM IR

This crate lives under `compiler/rustc_codegen_llvm`. It is responsible for lowering MIR to LLVM IR, two intermediate representations with sometimes subtly different semantics. One miscompilation (rust#111502) was *arguably* due to this crate, but ultimately it was fixed by changing `rustc_mir_transform`.

[3]https://github.com/rust-lang/rust/issues/79649

https://t.me/learningnets

	Total	Rustlantis Hit	Rustlantis Cov.	RustSmith Hit	RustSmith Cov.
Lines	13700	4219	30.8%	4189	30.6%
Functions	1211	388	32.0%	385	31.8%

Table 4.4: Coverage of `rustc_codegen_llvm`

LLVM Passes

LLVM has two types of passes: analysis passes, which compute information about the IR without mutating it, and transform passes, which modify the IR. Only transform passes can directly introduce errors, but since it often relies on information from analysis passes, bugs in either type of pass can result in miscompilation. rust#112061 and rust#112526 were due to bugs in these passes.

Analysis passes live under `llvm/lib/Analysis` directory of the LLVM Project repository

	Total	Rustlantis Hit	Rustlantis Cov.	RustSmith Hit	RustSmith Cov.
Lines	57800	25764	44.6%	23107	40.0%
Functions	3772	1826	48.4%	1714	45.4%
Branches	38438	19308	50.2%	17360	45.2%

Table 4.5: Coverage of LLVM Analysis passes

Transform passes live under `llvm/lib/Transform` directory of the LLVM Project repository

	Total	Rustlantis Hit	Rustlantis Cov.	RustSmith Hit	RustSmith Cov.
Lines	189316	54925	29.0%	48366	25.5%
Functions	11032	2979	27.0%	2806	25.4%
Branches	117258	40990	35.0%	36890	31.5%

Table 4.6: Coverage of LLVM Transform passes

Lowering to machine instructions

After passes on the IR, LLVM needs to lower LLVM IR into target-specific machine instructions. Again, this is not a simple, mechanical step as it

https://t.me/learningnets

involves steps like representing LLVM data types as native types with specific sizes, deciding on the ABIs of functions, and so on.

There are two steps in machine instruction lowering: a target-independent pass, and a target-specific one. The target-independent pass was responsible for rust#112170, and the x86-specific pass was responsible for llvm#63475.

Target-independent passes live under `llvm/lib/Codegen` directory of the LLVM Project repository.

		Rustlantis		RustSmith	
	Total	**Hit**	**Cov.**	**Hit**	**Cov.**
Lines	189276	57980	30.6%	54277	28.7%
Functions	9594	3529	36.8%	3358	35.0%
Branches	114890	43904	38.2%	41460	36.1%

Table 4.7: Coverage of LLVM target-independent codegen

Target-specific passes live under `llvm/lib/Target` directory of the LLVM Project repository. We only performed the instrumentation on x86-64 architecture, so we only have data for `llvm/lib/Target/X86`.

		Rustlantis		RustSmith	
	Total	**Hit**	**Cov.**	**Hit**	**Cov.**
Lines	93589	24454	26.1%	21627	23.1%
Functions	4043	1459	36.1%	1372	33.9%
Branches	76894	26112	34.0%	23724	30.9%

Table 4.8: Coverage of LLVM X86-specific codegen

Discussion

The LLVM coverage percentage is meaningful for comparison only. It is not possible to reach 100% coverage by generating Rust programs, as Rust cannot produce all possible forms of IR.

100% coverage on MIR optimisations is also impossible, as some MIR passes (such as `EarlyOtherwiseBranch`) have been disabled due to known correctness issues, but the code remains.

Compared to RustSmith, Rustlantis has around 4 p.p. lower coverage of Rust's MIR optimisation, marginally higher coverage of Rust's LLVM IR generator, around 4 p.p. higher coverage of LLVM's passes, and around 2 p.p. higher coverage of LLVM's machine code generator. The difference in

https://t.me/learningnets

coverage is not significant, and does not reflect the difference in the fuzzers' effectiveness: Rustlantis encountered 15 bugs (2 were previously known) on the most recent Rust versions; RustSmith encountered 5 (all previously known) which also fuzzed historic Rust versions.

We conclude that compiler coverage is not a very good predictor for the effectiveness of a fuzzer in terms of its ability to find bugs, especially miscompilation bugs.

4.4 Performance

Figure 4.2 shows the fuzzing (both generation and execution) throughput on an AMD EPYC™ 7742 processor with 48 of its physical cores made available to Rustlantis.

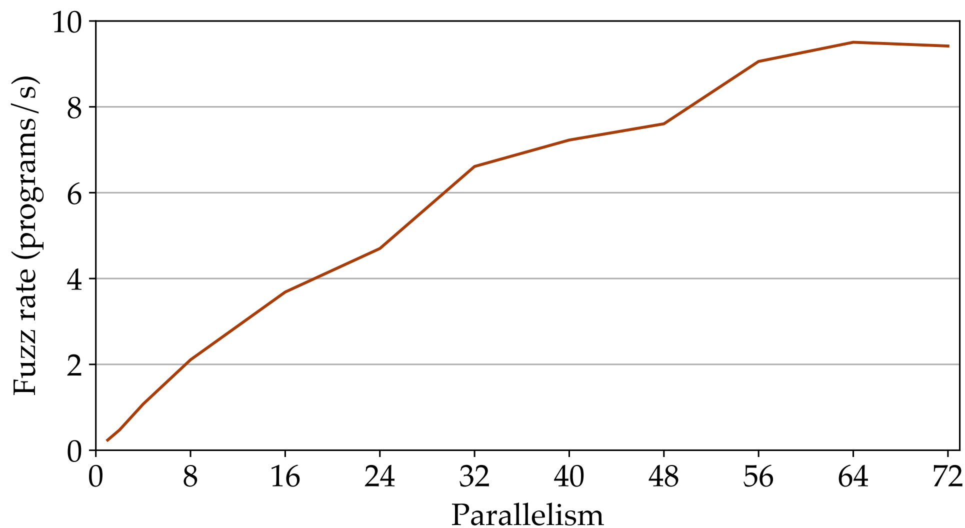


Figure 4.2: Fuzz rate by parallelism

As expected, the throughput increases linearly with the number of parallel instances up to a plateau when it is using all available CPUs. But the plateau did not start until 56 parallel instances, slightly higher than the number of available cores. This is likely because the program generation, compilation, or execution of the compiled program does not fully utilise a core all the time, it is therefore more efficient to overcommit CPUs so other instances can fill in the idle gaps.

Figure 4.3 shows where time was spent on programs of different sizes. The timings include both compilation and execution for the compiler-based testing backends.

https://t.me/learningnets

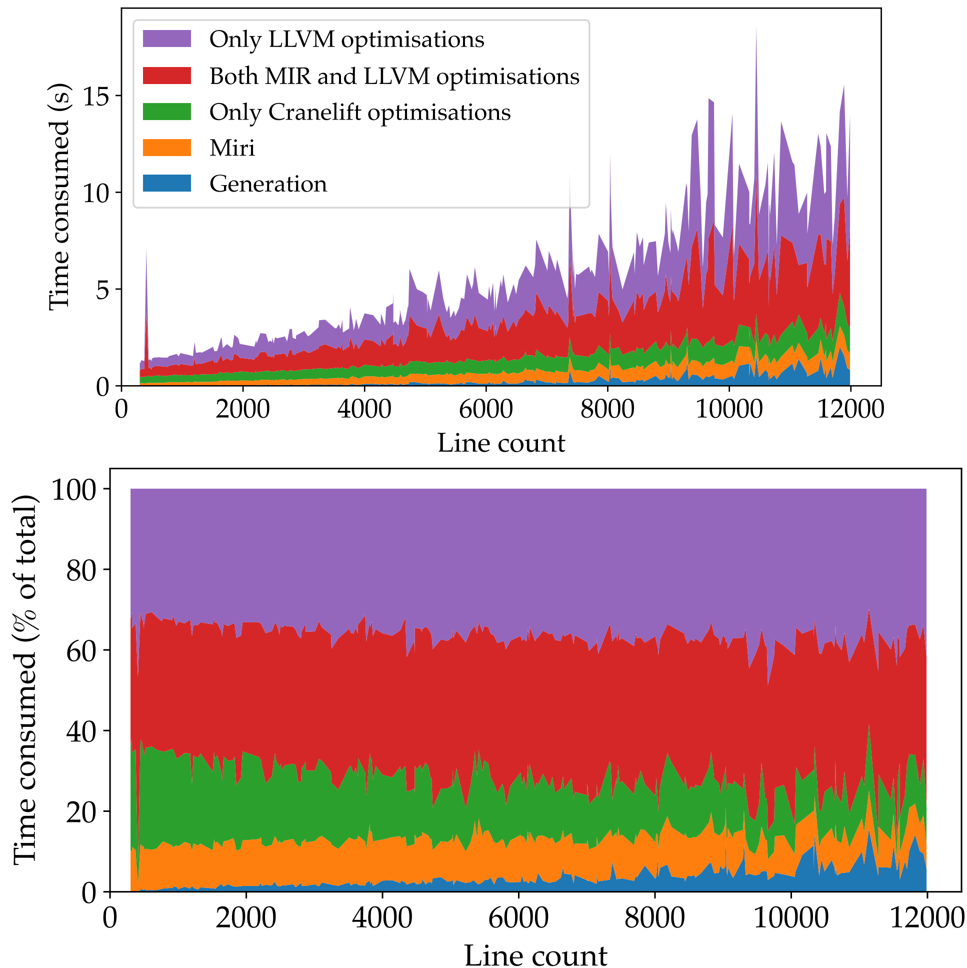


Figure 4.3: Fuzz time breakdown by line count

We can see that Rustlantis' program generation consumes very little time – less than 5% of the total time for programs below 8000 lines, which is the vast majority.

Interestingly, Miri the interpreter is the fastest testing backend, despite having a general reputation of being slow. Miri almost certainly is slower at *execution* when compared to a compiled binary, but it has the advantage of not having to compile or optimise anything, which is rather slow in LLVM.

https://t.me/learningnets

4.5 Energy efficiency

The machines we used did not record the energy consumption of our fuzzing jobs, but we can estimate this.

We will assume all our Euler jobs are done on nodes with two sockets of AMD EPYC™ 7742 processors, the most common node type in the Euler cluster. Two sockets (128 cores) of this CPU consume 240 W on average during Linux kernel compilation [20]. Our fuzzing jobs ran for about 394 hours on Euler, usually using 96 cores in parallel. Assuming the power input is evenly distributed on all CPU cores, our fuzzing job consumes 180 W of power on the CPU, which totalled 255 MJ, or 71 kWh of electrical energy.

The fuzzer also ran on an M1 Mac mini for about 96 hours, which has a peak CPU power consumption of 39 W [21]. This meant that it consumed at most 13 MJ, or 3.7 kWh of electrical energy.

Altogether, 4.5 CPU years of fuzzing consumed **75 kWh of electricity**, which is around 4 days' worth of electricity consumption per capita in Switzerland [22]. Note that this only accounts for the energy consumed by the CPU packages; other parts of the Euler cluster, such as the cooling system, certainly also consumed a notable amount of energy, but this is difficult to estimate.

https://t.me/learningnets

Chapter 5

Future Work

5.1 Missing language constructs

Rustlantis does not generate some common constructs in Rust, such as

- References
- Enums
- Unions
- Recursive types
- Heap-allocated (Boxed) types
- Zero-sized types
- Manually dropped types
- Non-`#[repr(rust)]` types
- ... and many more

Supporting these constructs would greatly increase the coverage of compiler code and potentially discover more bugs. For instance, bugs related to the `noalias` LLVM attribute mentioned in the first chapter can only be triggered with `&mut` function parameters.

But supporting these constructs imposes many challenges. For instance, the addition of references means that we have to maintain aliasing constraints imposed by the Tree Borrows model. Allowing struct, enum, and union definitions to refer to themselves makes it possible to create self-referential values. This is certainly interesting to generate but it also makes `PlaceTable` cyclic, requiring our graph walk algorithms to take this into account.

https://t.me/learningnets

5.2 Program reduction

Most Rustlantis-generated programs are between 3000-7000 lines long (Figure 5.1). This is too long to be a useful bug report for Rust maintainers. As a result, if a generated program triggers a bug, it must be reduced to a far smaller *minimal complete verifiable example* (MCVE) to be submitted as a bug report.

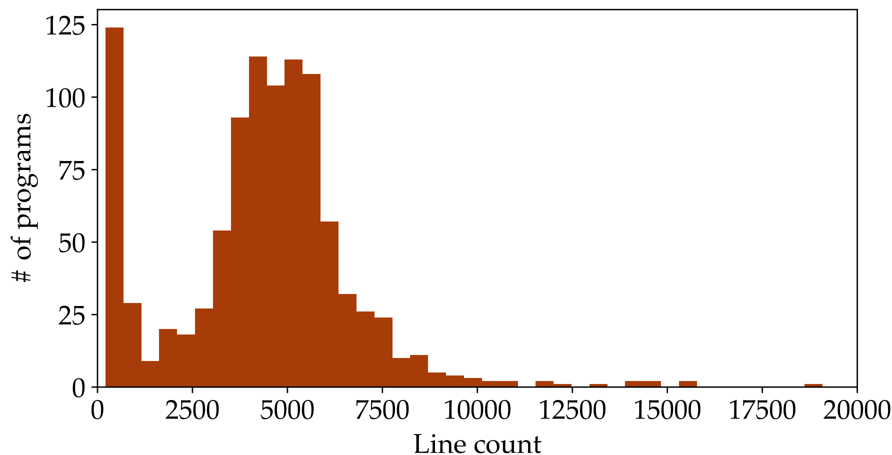


Figure 5.1: Size distribution of 1000 random programs

This problem is known as *test case reduction*. For the bugs discovered by Rustlantis, we 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

```
function reduce
    for line in line_count(prog)-1 to 0
        comment_out(line)
        result = differential_test(prog)
        if not has_bug(result) or has_ub(result)
            uncomment(line)
        else
            remove(line)
        end if
    end for
end function
```

This script is naive and inefficient. Many sophisticated techniques have been proposed in prior research [23], but we did not have time to adopt them to MIR programs.

https://t.me/learningnets

When the bug is in LLVM and we were able to isolate a reproduction in LLVM IR form, we used `llvm-reduce` [15] tool to reduce the size of the IR code. However, we found that this tool tends to introduce new UB during reduction and thus invalidates the test program, so it cannot be fully relied on.

5.3 Keep Rustlantis running

Upon completion of this project, we will no longer have access to Euler or other HPC clusters. But Rustlantis has shown to be capable of finding bugs and there are certainly still more bugs to be found. It would be highly regrettable for Rustlantis to become a one-off academic project and fade into obscurity due to the lack of compute resources, forfeiting all its potential.

Fortunately, after reaching out to The Rust Foundation, they have expressed interest in providing compute resources to keep Rustlantis running through its Cloud Compute Program[1]. The details are yet fully determined, but we are confident that Rustlantis will be actively used, maintained, and continue to find new bugs and regressions in the future.

[1] https://foundation.rust-lang.org/news/2022-06-09-cloud-compute-program-update/

https://t.me/learningnets

Bibliography

[1] sirkib, *Tail recursion assumed to terminate? Rustc 'solves' the Collatz conjecture*, Social Media. [Online]. Available: https://web.archive.org/web/20210227112302/https://www.reddit.com/r/rust/comments/ltm4ko/tail_recursion_assumed_to_terminate_rustc_solves/.

[2] C. Lattner and V. Adve, 'LLVM: A compilation framework for lifelong program analysis and transformation,' in *CGO*, San Jose, CA, USA, Mar. 2004, pp. 75–88.

[3] M. Marcozzi, Q. Tang, A. F. Donaldson and C. Cadar, 'Compiler fuzzing: how much does it matter?' *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Article 155, 2019. DOI: 10.1145/3360581. [Online]. Available: https://doi.org/10.1145/3360581.

[4] X. Yang, Y. Chen, E. Eide and J. Regehr, *Finding and understanding bugs in C compilers*, Conference Paper, 2011. DOI: 10.1145/1993498.1993532. [Online]. Available: https://doi.org/10.1145/1993498.1993532.

[5] V. Livinskii, D. Babokin and J. Regehr, 'Random testing for C and C++ compilers with YARPGen,' *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Article 196, 2020. DOI: 10.1145/3428264. [Online]. Available: https://doi.org/10.1145/3428264.

[6] W. Howden, 'Theoretical and empirical studies of program testing,' *IEEE Transactions on Software Engineering*, vol. SE-4, no. 4, pp. 293–298, 1978. DOI: 10.1109/TSE.1978.231514.

[7] W. M. McKeeman, 'Differential Testing for Software,' *Digit. Tech. J.*, vol. 10, pp. 100–107, 1998.

[8] M. Sharma, 'Rustsmith a randomized program generator for rust,' Thesis, Imperial College London, 2022. [Online]. Available: https://www.imperial.ac.uk/media/imperial-college/faculty-of-engineering/computing/public/2122-ug-projects/2122-individual-

https://t.me/learningnets

projects/RustSmith---a-Randomized-Program-Generator-for-Rust.pdf.

[9] *Module std::intrinsics::mir*. [Online]. Available: https://doc.rust-lang.org/std/intrinsics/mir/index.html (visited on 30/06/2023).

[10] T. R. Project, *Miri*, https://github.com/rust-lang/miri.

[11] N. Villani, 'Tree Borrows,' M.S. thesis, ENS Paris-Saclay, 2023. [Online]. Available: https://github.com/Vanille-N/tree-borrows/blob/eeb44c2509a6fa3f6e55f4bd75f5fd416a576676/half/main.pdf.

[12] V. Le, M. Afshari and Z. Su, 'Compiler validation via equivalence modulo inputs,' in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 216–226, ISBN: 9781450327848. DOI: 10.1145/2594291.2594334. [Online]. Available: https://doi.org/10.1145/2594291.2594334.

[13] B. Alliance, *Cranelift Code Generator*, 2018. [Online]. Available: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

[14] O. Tange, 'GNU Parallel - The Command-Line Power Tool,' *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011. DOI: 10.5281/zenodo.16303. [Online]. Available: http://www.gnu.org/s/parallel.

[15] D. T. Ferrer, 'LLVM-Reduce for testcase reduction,' 2019 LLVM Developers' Meeting, 2019. [Online]. Available: https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech22.

[16] 'Ieee standard for floating-point arithmetic,' *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[17] R. Jung, H.-H. Dang, J. Kang and D. Dreyer, 'Stacked borrows: an aliasing model for Rust,' *Proc. ACM Program. Lang.*, vol. 4, no. POPL, Article 41, 2019. DOI: 10.1145/3371109. [Online]. Available: https://doi.org/10.1145/3371109.

[18] C. Gillot, *Issue comment*, 2023. [Online]. Available: https://github.com/rust-lang/rust/issues/110947#issuecomment-1527815558.

[19] M. Sharma, *Rust*, https://github.com/rustsmith/rust, 2022.

[20] M. Larabel, 'AMD EPYC 7003 "Milan" Linux Benchmarks - Superb Performance,' 2021. [Online]. Available: https://www.phoronix.com/review/epyc-7003-linux-perf/7 (visited on 28/06/2023).

[21] Apple, *Mac mini power consumption and thermal output (BTU) information*, Dataset, 2023. [Online]. Available: https://support.apple.com/en-gb/HT201897 (visited on 28/06/2023).

https://t.me/learningnets

[22] Presence Switzerland, *Energy – Facts and Figures*, 2023. [Online]. Available: https://www.eda.admin.ch/aboutswitzerland/en/home/wirtschaft/energie/energie---fakten-und-zahlen.html.

[23] J. Chen *et al.*, 'A Survey of Compiler Testing,' *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, ISSN: 0360-0300. DOI: 10.1145/3363562. [Online]. Available: https://doi.org/10.1145/3363562.

[24] M. S. Hecht and J. D. Ullman, 'Characterizations of reducible flow graphs,' *J. ACM*, vol. 21, no. 3, pp. 367–375, Jul. 1974, ISSN: 0004-5411. DOI: 10.1145/321832.321835. [Online]. Available: https://doi.org/10.1145/321832.321835.

https://t.me/learningnets

Appendix A

Proof of reducibility

To prove that Rustlantis-generated MIR is reducible, we first prove that any control flow graph satisfying a particular property is reducible. We then argue that Rustlantis-produced control flow graphs have this property by construction, thereby proving the reducibility of Rustlantis-generated MIR.

Given a control flow graph, we can number all basic blocks as bb_0, bb_1, ..., bb_n, where bb_0 is the entry block. We say that bb_i is *lesser* than bb_j if $i < j$, denoted as $bb_i < bb_j$, and *greater* if the other way round. We partition the basic blocks into two sets: R and D, defined as:

Definition 1 *A basic block is in R if and only if it has no successor, or has a successor greater than or equal to it. A basic block is in D if and only if it has only lesser successors.*

Remark 2 *This means that $bb_0 \in R$: it is the least block and cannot have lesser successors.*

Unless otherwise quantified, we use r to refer to an arbitrary R basic block, and d for an arbitrary D basic block. $bb_i \to bb_j$ reads "bb_i is a predecessor to bb_j in the CFG" (or equally, bb_j is a successor to bb_i), and $bb_i \to^* bb_j$ reads "a path from bb_i to bb_j in the CFG".

We define $\textsc{PrevR}(bb_i)$ as the greatest R basic block less than bb_i, and $\textsc{NextR}(bb_i)$ as the least R basic block greater than bb_i. $\textsc{PrevD}$ and $\textsc{NextD}$ are defined in the same way. $\textsc{Lp}(bb_i)$ denotes the least node in bb_i's predecessor set.

For a control flow graph G, we assume that there is a numbering and R/D partition scheme such that the following holds:

Property 3 The least predecessor of any non-entry basic block bb is the greatest R basic block less than bb ($\textsc{Lp}(bb) = \textsc{PrevR}(bb)$).

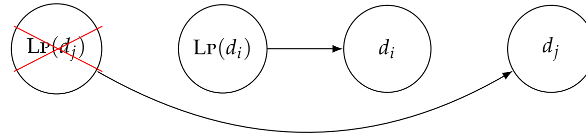
In other words, an edge cannot "skip" an R basic block.

https://t.me/learningnets

Corollary 4 *A D predecessor of bb_i is greater than bb_i*

This follows from Definition 1.

Theorem 5 *If $d_i < d_j$, then* $\text{Lp}(d_i) \leq \text{Lp}(d_j)$



Proof d_i is not bb_0 as $bb_0 \in R$, therefore $\text{Lp} = \text{PrevR}$ by Property 3. If $\text{Lp}(d_j) < \text{PrevR}(d_i)$, we have $\text{PrevR}(d_j) < \text{PrevR}(d_i) < d_i < d_j$, but this is impossible as there is another R block between $\text{PrevR}(d_j)$ and d_j. So $\text{Lp}(d_j) \geq \text{PrevR}(d_i)$ and $\text{Lp}(d_j) \geq \text{Lp}(d_i)$. □

Theorem 6 (Intermediate node theorem) *Given a path of basic blocks p from $p\langle 0\rangle$ to $p\langle n\rangle$, and a block x not in p where $p\langle 0\rangle < x < p\langle n\rangle$, there must be two adjacent blocks $p\langle i\rangle \to p\langle i+1\rangle$ such that $p\langle i\rangle < x < p\langle i+1\rangle$.*

Proof We can prove this by induction

Base case p has no block excluding $p\langle 0\rangle$ and $p\langle n\rangle$

$n = 1$, and we have $p\langle 0\rangle < x < p\langle 1\rangle$.

Inductive case Assuming there are $p\langle i\rangle < x < p\langle i+1\rangle$ when p has k blocks excluding $p\langle 0\rangle$ and $p\langle n\rangle$, to show that there are $p\langle i\rangle < x < p\langle i+1\rangle$ when p has $k+1$ blocks excluding $p\langle 0\rangle$ and $p\langle n\rangle$.

Since p has $k+1$ blocks excluding $p\langle 0\rangle$ and $p\langle n\rangle$, $p\langle 1\rangle$ exists. Either $p\langle 1\rangle > x$, or $p\langle 1\rangle < x$.

Case $p\langle 1\rangle > x$ we have $p\langle 0\rangle < x < p\langle 1\rangle$.

Case $p\langle 1\rangle < x$ there is a path from $p\langle 1\rangle$ to $p\langle n\rangle$ which has k blocks excluding $p\langle 1\rangle$ and $p\langle n\rangle$. Let this be p'. We have $p\langle 1\rangle = p'\langle 0\rangle < x < p'\langle n-1\rangle = p\langle n\rangle$. Using the inductive hypothesis, we have $p'\langle j\rangle < x < p'\langle j+1\rangle$, which means $p\langle j+1\rangle < x < p\langle j+2\rangle$ □

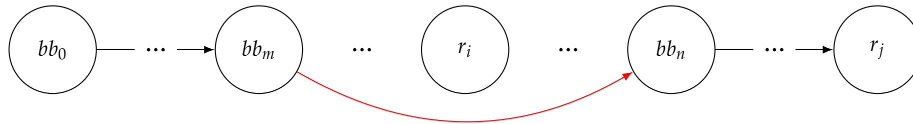
Theorem 7 (R dominates greater Rs) *If $r_i < r_j$, then r_i dominates r_j.*

Proof Assume there are $r_i < r_j$ where r_i does not dominate r_j.

This meant that there is a path from bb_0 to r_j which doesn't contain r_i.

We have $bb_0 < r_i < r_j$. Due to Theorem 6, there are two adjacent blocks $bb_m \to bb_n$ such that $bb_m < r_i < bb_n$.

https://t.me/learningnets



If $bb_m \in D$**,** this would contradict Definition 1, as its successor bb_n is greater.

If $bb_m \in R$**,** then bb_n would have an R predecessor smaller than r_i, which is an R known to be greater. Since $bb_n \neq bb_0$, this contradicts Property 3.

So a path from bb_0 to r_j without r_i is impossible. Thus r_i dominates r_j. □

Theorem 8 *The least predecessor of d dominates d*

Proof We prove this by strong induction over all D blocks from the end.

Base case To show that the least predecessor of the greatest D basic block d_{last} dominates it.

We have shown that an R basic block dominates all greater R blocks (Theorem 7), so $\text{Lp}(d_{last})$ (which is R) dominates all its other R predecessors, which are all greater.

If d_{last} were to have any D predecessor, it must be greater than d_{last} due to Corollary 4; but then d_{last} is no longer the greatest D block, so d_{last} has no D predecessor, only R ones.

Therefore $\text{Lp}(d_{last})$ dominates all predecessors of d_{last}, thus it dominates d_{last}

Inductive case Assume $\text{Lp}(d)$ dominates d for all $d_k \leq d < d_{last}$, to show that for $d_i = \text{PrevD}(d_k)$, $\text{Lp}(d_i)$ dominates d_i

As show in the base case, $\text{Lp}(d_i)$ dominates all R predecessors of d_i.

Consider an arbitrary D predecessor d of d_i. $d > d_i$ due to Corollary 4, therefore $d_k \leq d \leq d_{last}$. From the inductive hypothesis, we have $\text{Lp}(d)$ dominates d.

And since $d_i < d$, we have $\text{Lp}(d_i) < \text{Lp}(d)$ (Theorem 5). Least predecessors are R, thus $\text{Lp}(d_i)$ dominates $\text{Lp}(d)$ by Theorem 7. As the choice of d is arbitrary, we have $\text{Lp}(d_i)$ dominates all D predecessors of d_i.

Therefore $\text{Lp}(d_i)$ dominates all predecessors of d_i, thus it dominates d_i. □

Theorem 9 (*R* dominates greater *D*s) *If $r_i < d_j$, then r_i dominates d_j.*

Proof Either $\text{Lp}(d_j) < r_i$, $r_i = \text{Lp}(d_j)$, or $r_i < \text{Lp}(d_j)$

If Lp$(d_j) < r_i$**,** then $\text{Lp}(d_j) < r_i < d_j$, which means that there is an R block between $\text{Lp}(d_j)$ and d_j. Since $d_j \neq bb_0$, this contradicts Property 3, so this case is impossible.

https://t.me/learningnets

If $r_i = \mathbf{Lp}(d_j)$**,** then we have r_i dominates d_j from Theorem 8.

If $r_i < \mathbf{Lp}(d_j)$**,** then $r_i < \textsc{Lp}(d_j) < d_j$. We have r_i dominates $\textsc{Lp}(d_j)$ from Theorem 7, and $\textsc{Lp}(d_j)$ dominates d_j from Theorem 8. So r_i dominates d_j. □

Now we are ready to prove G is reducible.

Definition 10 *[24, Theorem 6] A control flow graph is reducible if all its edges can be partitioned into two sets: Forward edges and Backward edges, such that:*

- *Forward edges form a DAG with all basic blocks reachable from the entry block, and*
- *Backward edges are always from a block to its dominator.*

Theorem 11 *G is reducible*

Proof We first provide a scheme to partition edges in G into Forward and Backward edges:

- $r_i \to r_j$ where $r_i < r_j$ are Forward edges,
- $r_i \to r_j$ where $r_i \geq r_j$ are Backward edges,
- $r \to d$ are Forward edges,
- $d \to r$ are Backward edges, and
- $d \to d$ are Forward edges.

To show that Forward edges form a DAG with all basic blocks reachable from the entry block:

Lemma 12 *All R blocks are reachable from bb_0 using only Forward edges.*

Proof We can prove this by induction

Base case To show that $bb_0 \to \textsc{NextR}(bb_0)$

$bb_0 = \textsc{PrevR}(\textsc{NextR}(bb_0))$, by Property 3, bb_0 is a predecessor to $\textsc{NextR}(bb_0)$. This edge is a Forward edge in the form of $r_i \to r_j$ where $r_i < r_j$.

Inductive case Assuming $bb_0 \to^* bb_i$ contains only edges in the form of $r_i \to r_j$ where $r_i < r_j$, to show that $bb_i \to \textsc{NextR}(bb_i)$

$bb_i = \textsc{PrevR}(\textsc{NextR}(bb_i))$, by Property 3, bb_i is a predecessor to $\textsc{NextR}(bb_i)$. This edge is a Forward edge in the form of $r_i \to r_j$ where $r_i < r_j$. □

Lemma 13 *All D blocks are reachable from bb_0 using only Forward edges.*

https://t.me/learningnets

Proof By Property 3, all D basic blocks have an R predecessor, thus all D basic blocks can be reached from an R block. As we have proven in Lemma 12 that all Rs are reachable from bb_0, all Ds are subsequently reachable from bb_0. □

Lemma 14 *The Forward edges do not form a cycle*

Proof Assume there is a cycle. It either contains only R blocks, only D blocks, or a mix of both.

If the cycle contains R blocks only, then the cycle contains only edges in the form of $r_i \rightarrow r_j$ where $r_i < r_j$, which only produces paths with monotonically increasing blocks. The cycle is impossible.

If the cycle contains D blocks only, then the cycle contains only edges in the form of $d \rightarrow d$. By Definition 1, such paths have monotonically decreasing blocks. The cycle is impossible.

If the cycle contains both R and D blocks, we can pick an arbitrary R block in the cycle. To reach a D block then back to the original R block, one must have gone through $r \rightarrow d$ and $d \rightarrow r$ edges; similarly, we can pick an arbitrary D block in the cycle, to reach an R block then back to the original D block, one must have gone through $d \rightarrow r$ and $r \rightarrow d$ edges. But $d \rightarrow r$ is not a Forward edge, thus the cycle is impossible.

Therefore Forward edges cannot form a cycle. □

Lemma 15 *Backward edges are always from a block to its dominator*

Proof A Backward edge is either in the form of $r_i \rightarrow r_j$ where $r_i \geq r_j$, or $d \rightarrow r$.

For edges $r_i \rightarrow r_j$ where $r_i \geq r_j$, if $r_i = r_j$, then this is true as a basic block dominates itself. Otherwise, if $r_i < r_j$, we have r_i dominates r_j from Theorem 7

For edges $d \rightarrow r$, we know $d > r$ from Definition 1. Due to Theorem 9, r dominates d □

Lemma 12, 13, and 14 together satisfy the first property in the reducibility definition. Lemma 15 satisfies the second property. □

Theorem 16 *Rustlantis-generated MIR always has a numbering and R/D partitioning scheme such that Property 3 holds.*

Proof Rustlantis naturally has an order in which basic blocks are added. Under this order, the only time a basic block has **only** already-added (and therefore lesser) successors are decoy basic blocks added during the generation of a `SwitchInt` terminator. Since their contents are copies of an existing

https://t.me/learningnets

basic block, the terminator can only name an existing basic block. Therefore, Rustlantis' decoy basic blocks are D, and Rustlantis' actually executed (real) basic blocks are R.

When Rustlantis adds a new basic block, the only terminator that can reference (and therefore becomes a predecessor) the new block is the one Rustlantis currently producing, which is always a real basic block as Rustlantis never moves its generation cursor into a decoy block.

For `Goto` and `Call`, there are no blocks between the current and new real block, therefore the current block is the PrevR of the new one and a predecessor. And it will remain the least predecessor as Rustlantis will not go back and modify the terminator of existing blocks.

For `SwitchInt`, there can be new blocks between the current and the new real block, but the real block is always added last, so the blocks in-between can only be decoy. So the current block is the PrevR of the new one and a predecessor. And it will remain the least predecessor as Rustlantis will not go back and modify the terminator of existing blocks. ■

https://t.me/learningnets