Detecting Undefined Behavior Across Foreign Function Boundaries in Rust Programs

Julius de Jeu

TU Delft

Detecting Undefined Behavior Across Foreign Function Boundaries in Rust Programs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Julius de Jeu
born in Woerden, The Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture by Victoria de Jeu

Detecting Undefined Behavior Across Foreign Function Boundaries in Rust Programs

Author: Julius de Jeu
Student id: 4781775

Abstract

Memory access bugs exist in almost every compiled programming language. To solve this, modern programming languages like Rust use complex variable ownership systems that ensure memory safety. These kinds of ownership systems can be used to distinguish between variables that can only be read and ones that can be *safely* written to as well. These strict ownership rules are, however, limited when using external libraries. While the Foreign Function Interfaces (FFI) used to show what arguments external functions use can be written with these ownership requirements, enforcing them on the side of the external library is not required, potentially resulting in a discrepancy between what is expected on Rust's side and what happens in the external library.

To solve this problem we propose a novel mechanism for detecting memory safety violations across language boundaries. We implemented it in a tool called MiriPBT: a combination of MIRI, a tool that can enforce ownership rules at runtime, MiriLLI, an extension of MIRI that allows ownership rules to be enforced on the other side of the FFI boundary, and Property Based Testing, which allows us to greatly increase the size of the domain we can test. We use Rust's type system to generate inputs for the PBT and use the runtime checks of MIRI and MiriLLI to check if any ownership rules are violated. Finally we present the result of the PBT in a format an average Rust user can easily understand, helping them resolve any FFI related ownership bugs in their code.

Thesis Committee:

Chair:	Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. M.A. Costea, Faculty EEMCS, TU Delft
Committee Member:	Dr.ir. M.J.G. Olsthoorn, Faculty EEMCS, TU Delft

Preface

It has been quite a while since I started with Computer Science, starting in 2018 here in Delft with a bachelor's degree in Computer Science and Engineering. As it turns out, the "Engineering" part of the name implies a lot of math, my favorite subject... Anyways a year later I tried again, but this time in Utrecht with a bachelor's Informatica. This one was a lot more successful, even making it to the 'Honours Programme' there. For some reason I decided that going on an exchange was a good idea, so for half a year I lived in Oslo, Norway, where I learned a lot about Operating Systems, which I figured would be the focus of my thesis during my master's degree. However, as it turns out, I enjoyed programming language development a lot more than I had expected. After following a bachelor's course on it in Utrecht, I figured 'why not try the master's course in Delft?' This was a great idea as it turns out, since after 4 or so courses on PL development I wrote an entire thesis using one of my favorite programming languages, Rust.

This thesis highlights the end of my time as a student, which I (thankfully) did not spend only studying. My old student association from when I attempted my bachelor here in Delft quickly welcomed me back when I returned, confusing some people as to who I am and why I knew so many people there. As it turns out I liked AEGEE-Delft so much that I decided to do a board year there, even while following some courses for my masters here. In the end I can conclude that my time as a student was a lot of fun!

There are a few people I would like to thank as well: for starters Andreea and Jana, who helped me a lot with my thesis, Andreea more with the theoretical and writing part (and the patience to deal with Julius 'oh did I misunderstand that?' de Jeu), and Jana with the programming part. I also want to thank Jesper for giving me some nice comments on what to work on (and the deadline extensions you gave me, I'm pretty sure we're at 3 of them but I've lost count). Obviously my parents, who I have yet to torture with an entire weekend of presenting to, my sister who made the beautiful cover art and helped check my spelling and grammar (somehow this works for team dyslexia). Also a big thanks to the other people that checked my thesis for any issues. Finally a huge thanks to my friends from AEGEE, from my study and all other people that I've met throughout the years, I'm sure you've made my life a whole lot more fun!

Julius de Jeu
Delft, the Netherlands
September 4, 2025

Contents

List of Figures

List of Tables

Acronyms

crate Rust package

FFI foreign function interface

HIR high-level intermediate representation

IR intermediate representation

LE Local Execution

MIR mid-level intermediate representation

MIRI mid-level intermediate representation interpreter

MiriLLI MIRI extended with LLVM Bytecode interpretation

MiriPBT MIRI with PBT support

PBT Property-Based Testing

SP structure provider

Chapter 1

Introduction

Rust is a widely loved type safe language that uses ownership and borrowing to manage memory (Matsakis and Klock 2014; The Rust Programming Language Contributors 2025c; Shanker 2018; Stack Overflow 2022; Stack Overflow 2023; Stack Overflow 2024; Stack Overflow 2025). Ownership refers to the concept that each value in Rust has a single owner that is responsible for its memory. Borrowing allows references to owned values to be passed around. This combination lets Rust manage memory without a garbage collector, allowing it to be used on both operating systems and embedded systems, creating a replacement for C with guarantees of memory safety.

Unsafe Rust. To interact with code that does not follow Rust's guarantees, we will need to disable them for a short while. To do this we use `unsafe` blocks (Evans, Campbell, and Soffa 2020; Astrauskas et al. 2020; Jung 2016; The Fuchsia Developers 2024). While these blocks do not allow us to get complete freedom from the requirements that Rust has for its code, it does allow us to manipulate raw pointers. These raw pointers are effectively Rust's version of C's pointers, for example, creating a pointer to unallocated memory can be done with these raw pointers. One example where these raw pointers are required is in embedded programming. Rust supports more and more embedded platforms, but it is commonly needed to interact with raw memory addresses (Sharma et al. 2024), which means that we need to use raw pointers. This requires Rust's raw pointers to 'escape' from any borrowing and ownership rules, which means that it is possible to get memory access errors.

Foreign function interfaces (FFIs). A significant amount of mature libraries are written in C, but Rust needs a way to interact with them. Since Rust cannot directly access C code, doing this directly is impossible, but it can access bindings exposed via FFIs (McCormack, Dougan, et al. 2024). These bindings take a form of a template, for a function it would just be the signature without a body. These bindings are almost always considered `unsafe`, since there is no guarantee about what memory is being manipulated inside of the C function that is being called. This does, however, leave us with the problem that Rust cannot reason about raw pointers too well.

Stacked Borrows. Since Rust's own borrow checker (the part of the compiler that verifies that borrows are used safely) cannot reason about raw pointers, Stacked Borrows (Jung, Dang, et al. 2020) was created: an aliasing model for Rust that does allow us to track borrow usage across raw pointers. Since we need to trace every memory access, incorporating this into a regular binary is difficult. Doing this statically is also nearly impossible, because reasoning about raw pointers is equally challenging. Using a method that interprets Rust code would make the tracking a lot easier. Thus, an interpreter for Rust code is required.

MIRI. MIRI (Miri Contributors 2025) is an interpreter that uses Rust's mid-level intermediate representation (MIR) to symbolically execute Rust code. MIR is one of the intermediate steps that the Rust compiler uses to optimize the output of the compiler. At this level, the code will be divided into blocks, and each block can only be entered via a function call or

after a conditional such as a `if` statement. Since we can now trace execution at runtime, implementing a tool like the aliasing model provided by Stacked Borrows is now possible. There are, however, some limitations to MIRI: the first and most relevant is that it only works for Rust code since it is an interpreter for MIR. C code, for example, is not compiled to MIR, thus tracing execution across the FFI boundary is impossible. Next to that, MIRI is, due to its nature as an interpreter, quite a bit slower than a compiled program, since every MIR instruction has to be translated to one or more instruction(s) that the computer it is running on can understand. While it can be used to check specific tests, it is not suitable for large codebases. One other problem is that the output of any issues found with relation to Stacked Borrows are not user friendly. The output is understandable, but unlike normal Rust errors, it only points out that there is an issue and where the issue occurred, not how to solve the issue. Providing more information on what caused this issue would allow for safer code.

MiriLLI. Recent works have extended MIRI with LLVM bytecode interpretation functionality, resulting in MiriLLI (McCormack, Sunshine, and Aldrich 2025). Since MiriLLI is built on top of MIRI, it inherits the same issues as MIRI itself, such as the slowness of the interpreter and the lack of user-friendly output. It does, however, allow FFI calls to be traced as well, which allows it to check if the Rust code is following the rules of ownership and borrowing when calling into C code. For this it compiles the C code to LLVM bytecode, instead of a native binary, which allows it to use existing LLVM tooling to trace the execution of the C code the same way MIRI traces the execution of Rust code. MiriLLI now has access to both the memory of the Rust program and the memory of the C code, so it can check the rules of Stacked Borrows across the FFI boundary. Sadly, even if we can execute C code and check it via Stacked Borrows, we still have to deal with the limitations of the tests that we use. We cannot change the inputs to the tests in any way that allows us to find new bugs.

PBT. To solve the problem of limited tests, we will use Property-Based Testing (PBT) (Claessen and Hughes 2000). Normally, replacing a regular unit test with a property to test also requires computing the expected result. However, in our case the expected result is finding a Stacked Borrows violation, and as such we can simply ignore the result of the actual function and only check if we found a violation. Generating valid inputs for the function is done by using Rust's type system, it provides all the info we need to generate a simple structure of a function's parameters which we can then use to generate the inputs for the property. Since we are using MiriLLI, we can capture all calls to external functions, and then replace the arguments they have. This expands the search range of MiriLLI, allowing us to find more issues than just the ones that are in the test cases.

In this thesis we will provide a framework that can be used to check if a library called via FFI from Rust violates borrowing rules set by the structure of parameters. The tool will use MiriLLI to check the rules of Stacked Borrows, and will use PBT to generate the inputs to the function we want to test. Which still leaves us with two problems: can we improve the speed of MiriLLI and is it possible to generate a more user-friendly output.

Local Execution. The problem regarding the slow speed of the output we have solved by using a newly designed feature called Local Execution. It allows us to run the setup phase of MiriLLI only once, and then use PBT to vary the parameters to the function we want to test. This allows us to run the tests much faster, as we now do not have to re-run the setup phase for each test.

Friendly output. The user-friendly output problem will be solved by using PBT, but in a different way: instead of generating values, we will generate structures, altering the mutability of parameters and fields. This will allow us to report to the user what kind of mutability the parameters should have on the Rust side, solving the problem of the hard to understand output of MIRI and MiriLLI.

MiriPBT. Due to the use of PBT and MiriLLI, we will call the tool MiriPBT. MiriPBT will use a two-stage approach to testing: the first stage is called 'Detect', which will use PBT to generate values for the parameters of the function we want to test, finding violations of

the rules of Stacked Borrows. The second stage is called 'Debug', which will use the failing inputs of the previous step combined with PBT to generate structures for the parameters of the function we want to test, finding out which fields can be marked as immutable and which fields must be mutable, thus solving the problem of the hard to understand output of MIRI and MiriLLI.

Structure provider. The SP is the part of MiriPBT that generates the inputs for the PBT. It uses previously extracted type information to generate both values for parameters and structures. This generation of structures via PBT is a novel method that allows us to modify the mutability references and raw pointers in structures. Although the value generation does not benefit too much from an optimization like shrinking (Hughes 2007), the structure generation does benefit from it, since we want to minimize the changes required in the users code.

This thesis will turn the static type information that Rust provides into dynamic checks via PBT. The dynamic detection of type violations will happen via local testing, allowing a large speedup compared to current tools. To sum up, this thesis aims to address the following problem statement:

> *Can generating dynamic checks improve the ability to extensively test violations of Stacked Borrows in Rust code that calls into C code, and can this be used to generate output that allows the user to update their code to solve the violations?*

Contributions. The main contributions of this thesis will be the following:

A translation from types to dynamic checks via PBT.

In order to use our tool, we need to have the required type information to generate checks. Thus, we provide a method to translate types into dynamic checks. These dynamic checks depend on both the input function and the output property to test. These properties can range from a simple check to see if mutability rules are followed to more complex sets such as Stacked Borrows.

A novel method to generate PBT inputs from Rust's type information, both for values and structures.

We want to create a large variety of tests, a problem which PBT solves. By interpreting the type signature of a function we can reason both about values that parameters should have, and about the mutability that a signature in Rust provides. This system will implicitly support all forms of mutable references in Rust, both regular ones and raw pointers.

A modular framework that can be used to localize testing a specific property.

In order to speed up the development cycle, we want our framework to not slow down the user. MIRI and MiriLLI are interpreters, and while they are not too slow, waiting long on tests does not mean that more are found (Andrews et al. 2008). In order to not have to wait for a long time while running tests we attempt to localize test execution, only focusing on the function we are interested in.

Empirical evidence which shows that the method provides more information about the mutability of parameters than MIRI and MiriLLI do by using the violations of Stacked Borrows.

The Rust compiler normally provides good quality error messages (Zhu et al. 2022), but currently the output of MIRI is not up to date with regards to this. We will provide a solution

by using MiriPBT to generate a minimal example that can solve issues found, and present the user with a solution in steps on how to solve this, much like how the compiler does this for issues found.

Thesis Overview. The remainder of this thesis is split up into six chapters. Starting off with required background information, here the basis of this thesis will be explained in more detail. The next two chapters will explain the workings of our tool, first as a general overview, followed by a more detailed chapter. Afterwards, we will present our evaluation, followed by a chapter on related work to our thesis. Finally, we end with a conclusion.

The following list provides a more detailed overview of the chapters:

- In chapter 2 we will explain the required background, this includes information on the core technologies that have enabled us to write this thesis. These technologies range from the programming language Rust to Property-Based Testing.
- In chapter 3 we present an overview of the workings of MiriPBT, and give an example to show how MiriPBT works. This chapter also contains an introduction to the intricacies that our tool has, and shows that while it can work on any Rust code, why specifically C code is interesting to us.
- In chapter 4 we present a more detailed overview of our method, show the stages that our tool has to go through to help an end user make their code safer, explain why we decided on certain features for our tool and show how the code that is provided transforms.
- In chapter 5 we show how we evaluated MiriPBT, the research questions we answered, how we acquired our dataset and what the results that we present mean. Additionally, we discuss a list of issues which we had with MiriPBT.
- In chapter 6 we show related works to this thesis, building on top of this research to create our tool. We will show what the differences are between our work and the related works, and how we can use these works to improve our own.
- Finally in chapter 7 we conclude the thesis, we will provide a summary of our contributions and give suggestions for possible future work.

Chapter 2

Background

In this chapter, we will give an overview of the background required to fully understand this thesis. We will start by familiarizing ourselves with the programming language used for this thesis, Rust, and going into the relevant semantics that the language has. Next, we will look into an alternative to Rust's strict memory management scheme, that is to say, a system that works at runtime instead of at compile time. We will then continue by explaining the functionality of an interpreter for Rust, which is expanded upon in the section that follows with access to code written in other languages. Finally, we will end with a section on Property-Based Testing, which will form the foundation for the solution our thesis provides for the problem statement provided in the previous chapter.

2.1 Rust

In this section, we will talk about Rust's ownership semantics. We will start off by introducing the difference between Copy Semantics and Move Semantics. These two different models are two of the options that can enforce how variables can be used in programming languages, with Copy Semantics being more common, but Move Semantics being more resistant to memory management issues. Next, we will look into ownership, Rust's model that verifies that memory accesses are safe and valid, and do not result in undefined behavior. Next, we will look into accessing memory via references, focusing on Rust's specific method that it uses to ensure memory accesses stay safe. Next, we will focus on the part of the Rust compiler that enforces the rules of ownership and references, the Borrow Checker. This part of the compiler is the reason why we know that what Rust considers safe is actually safe. Finally, we will show what the difference between safe and unsafe Rust means, since these two are both required for most Rust programs that use FFI. This will be used as the basis of our explanation of Stacked Borrows, a slightly adjusted ownership model for Rust that supports raw pointers.

2.1.1 Move Semantics

In programming languages, variables can be handled in two different ways: via copy semantics or move semantics.

If a variable has Copy Semantics, it will be copied every time it is assigned to a new variable. This is quite efficient with smaller types, like integers or characters, but once we have a type with a pointer to a buffer inside of it, like a string) or a larger type, like the configuration for a program, copying all that data is inefficient.

This is where Move Semantics comes in, instead of copying the value, a special move operation is executed, invalidating the previous variable. This is commonly applied to more complex data types, like a vector or a string.

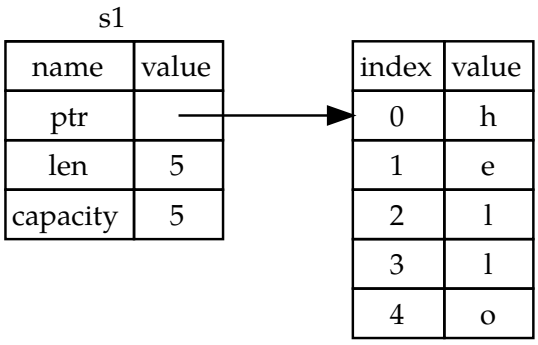

Figure 2.1: Initial state (The Rust Programming Language Contributors 2025d)

Commonly in Rust, values have move semantics, but there are some exceptions: if the used type implements the `Copy` trait, it will be copied instead of moved. This is the case for most primitive types, like integers and characters, but can also be implemented for custom types.

To explain move semantics with an example, we will use the following code:

```
fn main() {
  let s1 = String::from("hello");

  let s2 = s1;

  println!("{}, world!", s1);
}
```

which in memory looks like Figure 2.1. Once we try to copy `s1` into `s2`, we do not copy the data, but only the pointer to the data, which is shown in Figure 2.2. This is unlike copy semantics, where it would copy the data as well, which is shown in Figure 2.3. Finally, if we try to deallocate `s1`, we do not lose access to the data via `s2`, since it is still used somewhere, as shown in Figure 2.4.

2.1.2 Ownership

Rust has a unique approach to memory management, called 'ownership' (The Rust Programming Language Contributors 2025d). In Rust, every value has a single owner. These owners of values are scoped to a specific piece of Rust code, for example, a function body, or the body of an if-statement. If the scope is exited, there is no way to access the value anymore, and the memory used by the value is deallocated. This happens, for example, when a function returns. This relatively simple model allows Rust to guarantee memory safety without a garbage collector.

Here we present a simple example of Rust's ownership:

```
fn main() {
  let x = String::from("Hello, world!");
  let y = x;
  // println!("{}", x);
  println!("{}", y);
}
```

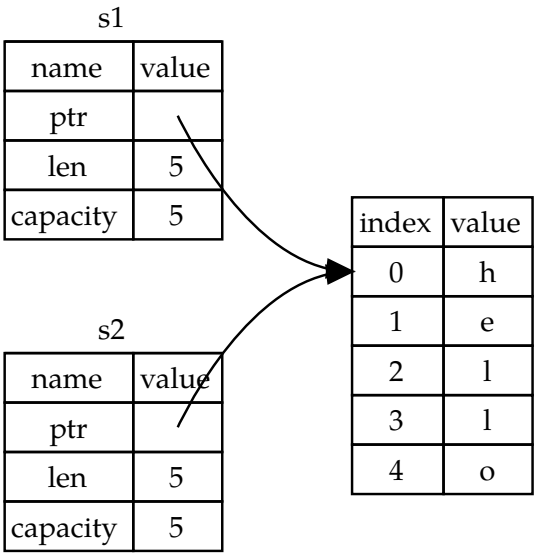

Figure 2.2: Move Semantics: we only copy the pointer, length and capacity (The Rust Programming Language Contributors 2025d)

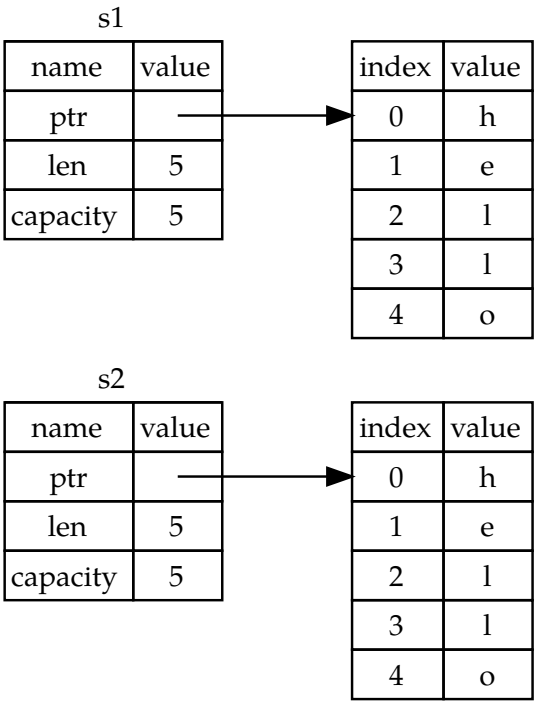

Figure 2.3: Copy Semantics: we copy the data as well as the pointer, length and capacity (The Rust Programming Language Contributors 2025d)

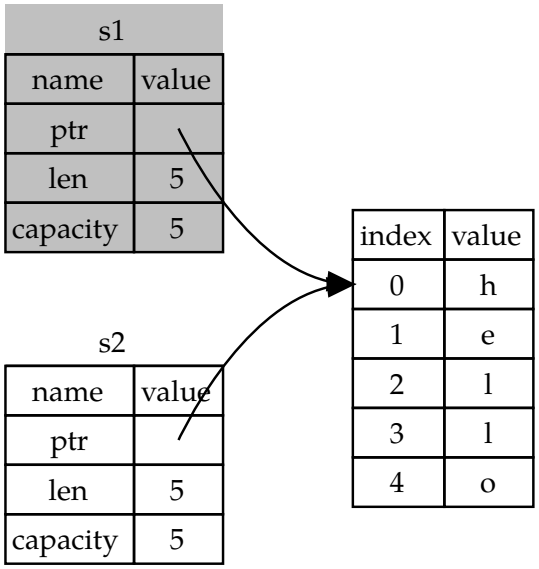

Figure 2.4: Move Semantics: after deallocating s1 (The Rust Programming Language Contributors 2025d)

In line 2 above, x was defined as the owner of the `String` we just allocated. In line 3 we move the ownership of the string from `x` to `y`, so we cannot use line 4 anymore, since `x` no longer is the owner of a value. The `println!()` call in line 5 does work however, since `y` is the owner of the `String`.

Furthermore, here is an example of moving values to a different function:

```
1  fn main() {
2    let x = String::from("Hello, world!");
3    print_string(x);
4    // println!("{}", x);
5  }
6
7  fn print_string(s: String) {
8    println!("{}", s);
9  }
```

The call on line 2 moves the variable `x` to the function `print_string`, meaning the `println!()` call on line 3 will not work, since `x` is no longer the owner of the variable. On line 7 we can see that we take a `String` as an argument to the function, thus we do the same reassignment as we did in the previous example, only now as part of a function call.

2.1.3 Borrowing

Only being able to move values around is not very useful, so Rust also allows borrowing (The Rust Programming Language Contributors 2025a). Borrowing is the process of creating a reference to a value without taking ownership of it. These references can be either mutable or immutable.

Here is an example of borrowing in Rust:

```
1  fn main() {
2    let x = String::from("Hello, world!");
3    let y = &x;
```

```
4     println!("{}", y);
5     println!("{}", x);
6   }
```

On line 3 we now borrow the `String`, instead of moving it. Since multiple immutable borrows are allowed we can now reference the `String` via both `y` and `x`, but do keep in mind that when we use `x`, we also remove any access via `y`.

Here is an example of mutable borrowing in Rust:

```
1   fn main() {
2     let mut x = String::from("Hello, world!");
3     let y = &mut x;
4     y.push_str(" How are you?");
5     println!("{}", y);
6     println!("{}", x);
7   }
```

Rust's borrowing rules are quite strict when it comes to which variables can modify which values. In line 3 above we create a mutable borrow of `x`. Unlike the immutable borrow from the previous example, we can modify the underlying `String` with a mutable borrow. This is shown in line 4, where we append another string. Since a mutable borrow can also act as an immutable borrow, we can still read it in line 5 and 6.

Lifetimes

Though not required to understand this thesis, lifetimes are an important part of Rust's ownership model. Lifetimes are a way to track how long a reference is valid. If we were to draw the lifetimes of the first borrowing example, it would look like this:

```
1   fn main() {
2     let x = String::from("Hello!"); // ----------+-- 'a
3     let y = &x;                     // --+-- 'b  |
4     println!("{}", y);              //   |       |
5     println!("{}", x);              //   |       |
6                                     // --+       |
7   }                                 // ----------+
```

The lifetime `'a` must outlive `'b`, shown as `'a: 'b` in Rust. This is fortunately rather obvious, since we need the reference to `x` to exist for a shorter amount of time than `x` itself, otherwise we could not have a valid reference.

2.1.4 Borrow Checker

The Borrow Checker (The Rust Programming Language Contributors 2025b) is a fundamental part of the Rust compiler, being the part that governs memory safety by verifying ownership and lifetimes. It verifies the requirements presented in the previous sections statically at compile time. Since it operates at compile time, it helps remove the need for a garbage collector, if it is known beforehand where a value should be de-allocated there is no need to check if memory is unused. Running at compile time makes it so the following classes of memory safety issues can be completely eliminated:

- Use after free issues: Since it is known when a value is de-allocated, it is impossible to use it after it has been freed.

- Dangling pointers: Since references have lifetimes, and those lifetimes are bound to the lifetime of the owning value, it becomes impossible in safe Rust to create dangling pointers.
- Data Races: Since Rust is very explicit regarding what types can travel between threads, it becomes very difficult to accidentally create a data race in safe Rust.

While the borrow tracker guarantees that Rust has memory safety, there is no guarantee on memory liveliness, so memory leaks are still possible. This might, for example, happen via `Box::leak`, which is meant to move the value of a box to a raw pointer, which is sometimes needed for interacting with C code.

2.1.5 Safe and Unsafe Rust

The ownership model, borrowing and Borrow Checker only apply to safe Rust. An average programmer will be able to do all they want with safe Rust. Unsafe Rust, on the other hand, relaxes some requirements set by the Borrow Checker. Code needs to be marked with an `unsafe` keyword to show that the Borrow Checker has to relax its requirements here. Using unsafe Rust allows the following extra features compared to safe Rust:

- Dereferencing raw pointers: Rust normally only allows the creation of raw pointers in safe code, but to use them unsafe code is needed.
- Calling unsafe methods: Unsafe methods are methods that require an invariant before they can be called, but the compiler cannot reason about this invariant on its own. Thus an unsafe marker is used to show that the programmer has to worry about this invariant. Another example of an unsafe method is one that interacts with C code over FFI.
- Accessing or modifying mutable static variables: Since static variables are global, in safe Rust it is only possible to access immutable static variables. The mutable variant always requires unsafe code, since the user has to worry about data races themselves.
- Implementing unsafe traits: unsafe traits work like unsafe methods, they require an invariant to be upheld when they are used. The `Send` and `Sync` traits are an example of unsafe traits: these two are needed to allow a value to be move to another thread, or move a reference to a value to another thread safely.
- Accessing union fields: Unions in Rust exist for C compatibility, since they are commonly used as a replacement for Rust's enums. However, since there is no way to know what representation of a union is the current one used, it is always unsafe to access fields in a union.

These extra features allow more freedom to a user, but at the same time it acts as a contract: instead of the Borrow Checker keeping track of things, the developer now has to do so.

2.2 Borrow Tracker

MIRI also supports a Borrow Tracker, which, in contrast to the Borrow Checker, enforces borrowing rules at runtime. The static analysis of the Borrow Checker is expanded upon by tracking the exact state of all variables during program execution. This enables it to detect violations of Rust's ownership model, and borrowing rules that may not be fully capable via the static Checker alone, for example, issues involving unsafe code.

Since the Borrow Tracker is working at runtime, it is often more strict than the compile-time Borrow Checker: this happens since it can now reason about raw pointers as well as

regular references. It can reject programs that compile due to issues with unsound behavior that can lead to subtle violations of the ownership model. This strictness is intentional as the Borrow Tracker is not meant to enforce that the program is valid Rust: that is the purpose of the Borrow Checker. Instead, it is meant to show undefined behavior in corner cases, especially when involving unsafe code.

Currently, MIRI provides two implementations of a Borrow Tracker:

- **Stacked Borrows** (subsection 2.2.1): a model that represents borrows as a stack. New borrows are pushed on top of a stack, and on access the state of the stack is checked to be in a valid state. If not, undefined behavior was detected. Currently this model is the default in MIRI.
- **Tree Borrows** (subsection 2.2.2): Tree borrows are a new alternative to stacked borrows: instead of using a stack, it uses a tree of borrows. This allows it to be more permissive than the current Borrow Checker is regarding when variables are used.

In our thesis we have focused on the Stacked Borrows implementation primarily. This is motivated by two factors: it is the default Borrow Tracker used by MIRI, and it is more complete than Tree Borrows.

2.2.1 Stacked Borrows

Stacked Borrows (Jung, Dang, et al. 2020) is a proposed system for borrow management that is more fine-grained than the current system in Rust. As the name suggests, borrows are placed on a stack, which is used to track the validity of references. Each owned value has a stack of borrows, which is used to track the validity of references to that value. This stack is then used on every access to the value (or a reference to it) to check if the access is valid.

Stacked Borrows provides a set of rules to follow, which guarantee that the memory used is not accessed in any way that can cause issues. We will start by showing some examples of how the system works, and then we will explain the rules that are enforced by the system. For every sample there will be a function `fn used(_: &mut i32)` or `fn used(_: &i32)` that takes a reference to an `i32` as an argument.

In the samples below we will comment out a section of code that if uncommented will cause a runtime error due to the stacked borrows system. The term *invalid* will be used to describe a borrow that is not active or is the wrong kind of borrow. It will be explained with an example below.

The system enforces a set of invariants that guarantee memory safety. In every example below we will include a comment after every line that shows the stack of all stacks of borrows.

- **Valid Mutable Borrow**: Regular mutable borrows follow the rules set by the Rust compiler, which means that it is not allowed to have multiple mutable borrows active at the same time. In line 9, we can see that `z` was removed from the stack, since we have accessed `y` which is after it. Note that while this code is valid under stacked borrows, currently it does not compile: Rust does not allow two mutable references at once, even if one gets invalidated before a different one is being used. If line 10 would not be a comment, Stacked Borrows would detect an issue, since it would require mutably borrowing `y` and `z` at the same time. This is, however, not allowed, since we cannot have two mutable references to a value at the same time. Reversing the `used(y)` and `used(z)` lines would allow the code to run normally however, because then we can drop the first reference after it is used.

```
1  fn valid_mutable_borrow() {
2    let mut x = 42;
3    // (42, [Unique(0)])
```

```
4     let y = &mut x;
5     // (42, [Unique(0), Unique(1)])
6     let z = &mut x;
7     // (42, [Unique(0), Unique(1), Unique(2)])
8     used(y);
9     // (42, [Unique(0), Unique(1)])
10    // used(z);
11  }
```

- **Raw Mutable Borrow**: This is where Stacked Borrows starts to differ from the regular borrow checker: raw mutable borrows are treated the same as regular mutable borrows. If this would not be the case, it would be possible to create multiple mutable references to the same value, which would lead to undefined behavior. If the code in line 6 were to be enabled, we would have to invalidate `y`, since it is not allowed to have two different raw mutable borrows: adding a new `SharedRW` to the stack requires that above it are no other `SharedRW` variables.

```
1   fn raw_mutable_borrow() {
2     let mut x = 42;
3     // (42, [Unique(0)])
4     let y = &mut x as *mut i32;
5     // (42, [Unique(0), SharedRW])
6     // let z = &mut x as *mut i32;
7     // (42, [Unique(0), SharedRW, SharedRW?])
8     used(unsafe { &mut *y });
9     // (42, [Unique(0), SharedRW])
10  }
```

- **Writing to an Immutable Borrow**: Changing an immutable borrow to a raw pointer, then to a mutable pointer and then writing to it is not allowed. Since the original borrow in the stack was immutable, the entire stack is immutable. This code runs normally when using the Borrow Checker, but when using Stacked Borrows line 8 produces an error: if there is an immutable pointer underneath a `SharedRW`, you cannot write to it.

```
1   fn writing_to_immutable_borrow() {
2     let x = 42;
3     // (42, [Shared(0)])
4     let y = &x as *const i32;
5     // (42, [Shared(0), SharedRO])
6     let z = y as *mut i32;
7     // (42, [Shared(0), SharedRO, SharedRW])
8     used(unsafe { &mut *z });
9   }
```

In addition to these examples, items that have interior mutability, like `RefCell` or `Mutex`, are also supported. These types allow for mutable access to the underlying data, while still following the rules of Stacked Borrows. These items all have a `UnsafeCell` inside of them, which allows for mutable access to the underlying data.

2.2.2 Tree Borrows

Tree borrows (Villani 2023) is an alternative to Stacked Borrows that uses a tree structure to track the validity of references. It is a more complex system, but it allows for more fine-grained control over the validity of references.

Tree borrows works by making a tree structure for every owned value, instead of the stack used by Stacked Borrows. This tree is used to track the validity of references. Each node in the tree represents a borrow, and the edges represent the relationships between the borrows. This means that it would be possible to have both a mutable and an immutable borrow active at the same time, as long as they are not related to each other.

This more permissible model allows for a more fine-grained control over the memory used, and also allows multiple currently incompatible borrows to exist at the same time. This results in a stricter model which safely allows code that currently is not allowed.

2.3 MIRI

MIRI (Miri Contributors 2025) is a tool that can be used to detect undefined behavior in Rust programs. It does this by interpreting Rust's MIR. This interpreter allows us to detect several issues that can only be found at runtime:

- Reading Uninitialized data: Raw pointers can be initialized to a wrong address, or null. Rust cannot easily check this itself, since the Borrow Checker cannot reason about most raw pointer problems.
- Out of bounds memory access: This often happens when the bounds on a List structure are set incorrectly. This, too, can only happen using Unsafe Rust, meaning that the Borrow Checker is not suited to solve this problem.
- Memory alignment issues: This is a rare issue that presents itself when types are converted between ABIs. When memory is misaligned some types will work slower or not at all, for example, when a `u32` is not aligned to 4 bytes.
- Invalid values for booleans or enums: While Rust allows converting between booleans or enums and integers, it is not guaranteed that a conversion in the other direction works as expected.
- Data races: since the Borrow Checker can only reason about code statically, it can only guarantee that the types used are allowed to change threads. It is, however, still possible for data races to occur when manually creating types that can travel between threads.

Since MIRI is an interpreter, it is not a replacement for running Rust normally. The execution is much slower compared to compiled Rust. Since it is an interpreter, however, it is easier to change the internal workings of it without having to recompile the entire compiler itself. Two examples of tools implemented in MIRI are Stacked Borrows and Tree Borrows, with both using the same framework that allows tracing memory access across the entire compiler.

Another issue that MIRI has, is that it can only use a single external library at a time. Since it has to load them at runtime, it cannot use statically compiled libraries and needs the dynamic version instead. Finally, the largest issue is that it is simply handing the variables used for an external call off to the library, and not tracing this itself. Thankfully, MiriLLI solves this problem.

Finally, a minor, but still relevant, problem that MIRI (and MiriLLI) experience is that the output is not always as clear as one might like it to be. This is especially the case when using Stacked Borrows or Tree Borrows. The Borrow Tracker framework is quite solid, but the output it produces is not very clear. Experienced programmers can understand what the problem is, but Rust's error style is meant to also support inexperienced programmers.

2.4 MiriLLI

MIRI is a great tool, but as explained above, its inability to talk to native libraries was quite limiting. This brought about MiriLLI (McCormack, Sunshine, and Aldrich 2025), which uses LLVM bytecode to trace memory access across the FFI boundary.

By compiling native libraries both to the regular output format and LLVM Bytecode, we can interpret that bytecode at runtime to trace memory access through it. If all calls to reading and writing to a specific memory address are captured we can reason about the code used in the 'native' library the same way we can reason about the code written in Rust. Combining these two items allows us to use borrow trackers to reason about C code, which presents the basis for our thesis. One important part is that since we are using a modified interpreter, most normal Rust code will still work. A limitation, however, is that all native code still has to go through both the LLVM interpreter and MiriLLI, which results in a slowdown compared to normal MIRI.

Another important piece of data provided by (McCormack, Sunshine, and Aldrich 2025) is the dataset they have used for their experiments. This dataset is useful to us, since we are basing our approach on MiriLLI. The most interesting part of this dataset for our approach is the tests that passed, since those did not show any undefined behavior.

2.5 Property-Based Testing

PBT (Claessen and Hughes 2000) is a testing technique that focuses on properties of the code, instead of specific test cases. This means that instead of writing a test case that checks if a function returns the correct value for a specific input, we write a property that should hold for all inputs.

As an example, we will use the following Rust function that adds two numbers:

```
fn add(a: i32, b: i32) -> i32 {
  a + b
}
```

Writing a single test case for this function is also quite simple:

```
#[test]
fn test_add() {
  assert_eq!(add(1,2), 3);
}
```

A property is a statement about the code that should always be true. For example, a property for a function that adds two natural numbers could be that the result is always greater than or equal to both of the input numbers. We could write this property as follows:

```
fn prop_add(a: u32, b: u32) -> bool {
  let result = add(a, b);
  result >= a && result >= b
}
```

Now that we have a property, we need to generate test cases that check if this property holds. This is done using generators, which are functions that generate random values of a specific type. For example, we could use the following generator to generate random natural numbers:

```
fn gen_u32() -> i32 {
  rand::random_range(0i32..)
}
```

which can then be used to generate random test cases for our property:

```
#[test]
fn test_prop_add() {
  for _ in 0..1000 {
    let a = gen_u32();
    let b = gen_u32();
    assert!(prop_add(a, b));
  }
}
```

Normally, these generators will try some specific values as well. For numbers they commonly try 0, 1, -1 (if the type supports it) and the maximum and minimum values for the type. This is done to ensure that edge cases are also tested.

In our example above, it would fail if the two numbers added together overflowed. This is a common issue in programming, and it is something that can be easily missed when writing test cases, but with PBT it is much more likely to be found.

We can also incorporate the result of a test case into the next one. This is called shrinking (Hughes 2007). It is done by altering the input range based on the previous result. For example, if we would find that adding two large numbers causes an overflow, we could shrink the input range to find exactly which numbers cause the overflow. This is done by generating smaller and smaller numbers until we find the smallest pair that causes the overflow.

Shrinking is used to find the smallest input that causes a failure (hence the name). This is useful for debugging, since it allows us to find the root cause of the issue. For numbers, this is quite simple: we can just halve the number until we find the smallest one that causes the issue. Vectors, Arrays and Maps are more complex, but the idea is the same: we try to remove elements from the collection until we find the smallest one that causes the issue. Rust's `enum` types are similarly complex, especially if they have special meanings, but shrinking them is still possible, as long as we have a default value to shrink towards. `Option` types are a good example of this: we can shrink `Some(x)` to `None`, and then shrink `x` further if needed.

Chapter 3

Informal Method

In this chapter we start by looking at an example of a program that can experience undefined behavior across the FFI boundary. Afterwards, we will show how we have fixed this problem using MiriPBT. For this we will look into a transformation of the source code into a concrete data type that we can use to generate values for a PBT, and how the code is executed.

The code below is split into two parts: one part Rust, and one part C. The Rust code calls the C code via a FFI from the `unsafe` block. The function that is called has an immutable raw pointer and a boolean flag. While the Rust code shows that our raw pointer is immutable, the C code is not aware of this, allowing us to assign to the immutable raw pointer in the C code.

```rust
fn main() {
    let x = 10;
    let b = false;
    unsafe {
        run(&x, b);
    }
    println!("{}", x);
}
#[miripbt_macros::marker_extern]
unsafe extern "C" {
    fn run(i: *const i32, flag: bool);
}
```

with the following C library also compiled into the executable:

```c
void run(int32_t *i, bool flag) {
    if (flag) {
        *i = 41;
        printf("Magic?\n");
    } else {
        printf("No magic %d\n", *i);
    }
}
```

If we run this program in its current state, it will first print the string 'No magic 10', followed by 10. If we were to change the value of `b` to `true` however, it would print 'Magic?' followed by 41. As shown in subsection 2.1.3, we should not be able to modify an immutable reference, but we did manage to do so since C does not follow Rust's ownership model.

This presents our first issue: we need to have full coverage of all arguments to our run function. In this example, it is quite easy to at least make a test where `flag` is `true` and `false`,

but for larger projects this may become quite difficult. Finding a solution to this for larger projects may make testing easier, even for smaller projects.

3.1 Detect

One solution to this problem could be that we want to use property testing, where we instead focus on defining a property that our function must follow. The property we want to uphold is in our case not violating Stacked Borrows (Jung, Dang, et al. 2020), and to test this we need to generate inputs to the functions we want to test.

From this we generate our PBT, based upon the function signature and the expected outcome of our test. The main types that are relevant for us are the function arguments, so in the case of our example above a `*const i32`, a raw pointer to a 32-bit integer, and `bool`, a simple boolean. We want to focus on whether or not any violations are found with Stacked Borrows. This also means that we can exit early if we find any issues with stacked borrows. It is quite difficult to write this property as a regular Rust function, since we depend on the environment (in our case MiriPBT) finding the issue during function execution. To represent this as a property, we have to incorporate the Borrow Tracker into our test. Note that this is not how the code actually works, it is just used as an example. The `with_sb` function takes a closure that has it's inner workings verified with Stacked Borrows, returning `false` if it found a violation, and `true` if it did not. Implementing this is possible using MIRI however, since it allows us to enable and disable Stacked Borrows at runtime.

```
#[test]
fn test_run() {
    for _ in 0..10 {
        assert!(with_sb(|| {
            let a = gen_i32();
            let b = gen_bool();
            unsafe { run(&a, b) };
        }));
    }
}
```

Since we have marked the `run` function, we will have MiriPBT generate type information for it. The `json` output of our tool contains the function name and the argument types. The `type` field can have five different values:

- Value: a normal value. This implies that we are not dealing with a reference. If this is the case, we know that this is the value we want to adjust.
- RawConst and RawMut: A raw immutable and mutable pointer. These values show us that we are working with raw pointers. While for us a pointer and a reference are the same, it is still important to show the distinction in case we want to focus on references or pointers specifically.
- Ref and RefMut: An immutable and mutable reference. These are the safe version of raw pointers and explained in subsection 2.1.3. For MIRI and MiriLLI there is no difference between these and the raw values, since they are both simply considered a 'pointer', with mutability handled at a different level.

```
{
    "run": {
        "args": {
```

```
            "i": {
                "type": "RawConst",
                "value": {
                    "type": "Value",
                    "value": "i32"
                }
            },
            "flag": {
                "type": "Value",
                "value": "bool"
            }
        }
    }
}
```

Using this format we can find inputs that cause issues. These inputs are provided by a separate program, the structure provider. Using the list of functions and a list of other non-primitive types that appear in the signatures it can generate valid input for a specific function. This input is then used by MiriPBT to see if our property is violated or not.

3.2 Debug

Once we find out that an input exists that has violated our property, we can get into the next stage where we try to analyze what the cause of the violation was. For this analysis we employ Shrinking to reduce the amount of changes needed in the user's code. For example, here it can only alter the mutability of `i`, resulting in the following signature:

```
fn run(i: *mut i32, flag: bool)
```

Using this signature shows that the function follows our property again, meaning it is safe to use and does not cause undefined behavior.

This is a very simple example, but it shows the general idea of what MiriPBT does. It finds inputs that violate a property, and then tries to find the minimal change needed to make the function follow the property again. This can be very useful for larger projects where it may be difficult to find all possible inputs that can cause issues.

3.3 Local Execution

While being able to verify properties like this improves code quality, it also increases the time required to run tests, since instead of starting it once, it needs to restart the test multiple times to see if it can prove the property always holds. To solve this problem, instead of executing the entire test every time, we focus only on the function we want to test. Considering the execution of this function is focused on a smaller local part of the program, we call this Local Execution.

Using Local Execution we can speed up our tests quite a bit, since we no longer have to run any setup code when we want to test new values. This is especially useful when the setup code is large, and the function we want to test is small.

To solve this problem we hook into MiriLLI's execution engine, and take over execution when we reach a function we want to test. This means we start in a valid state to enter the function, and we can just focus on executing the function itself.

Once we swap the original arguments for the ones we have generated, we can continue execution until we reach the end of the function. If we reach the end without any issues,

we can conclude that the property holds for this input. If we instead reach an error, we can conclude that the property does not hold for this input.

Chapter 4

Method

In this chapter we will share a more in-depth overview of how this thesis works. We will start by showing how our project is structured and what all the components are. Next, we will continue with our design choices, why we decided to use specific techniques, and how we split up the different tools we provide. Finally, we will focus on explaining the various transformations we have to make to the source code of a Rust program to enable MiriPBT to work correctly.

In this thesis we propose a method to more easily test Rust's Borrow Checker, Stacked Borrows and Tree-based Borrows. For this process we will use PBT to generate inputs for pre-marked functions. Furthermore, we will generate structures using PBT, which can be used to alter the mutability of variables and fields at runtime. These types will then allow us to see if the code written can either be relaxed. For example, if a field of a struct is marked as mutable but never written to, that mutability can be removed. The opposite situation, when a field is marked as immutable but mutated, does not often occur in Rust, but it can happen when using FFIs from Rust. For code only used by Rust, we will use MIRI, but we will have to use MiriLLI for code that uses FFIs. This is because MIRI does not support tracing memory access across FFIs, while MiriLLI does.

4.1 Project

Figure 4.1 shows how MiriPBT works. As input it takes a Rust crate, which has its marked functions transformed (1) into a format that allows us to extract the structure, a random input, which is simply any test that executes the function we are interested in, arrow (2) marks the entry into the function. As a final input we have a property P to test. In our examples we have used 'Does not violate Stacked Borrows' as our property, but this can be replaced by a different property, such as a subset of Stacked Borrows. The 'Detect' stage has two different outputs. Either we follow (5) and find no violations of property P, or we continue on to 'Debug' via (6) if we did find violations of property P, for which we also need an entrypoint to the function we want to test (3) and the original property to test (4). Finally, using the output from 'Debug' we go via 7 to report the minimal amount of changes needed to make the code run safely.

4.1.1 Detect

The 'Detect' stage (Figure 4.2) of MiriPBT uses the value PBT to find values that cause MiriLLI to abort with errors regarding Stacked Borrows. This stage communicates with the SP to generate the values used, and potentially for shrinking as well. We will use the following code as an example:

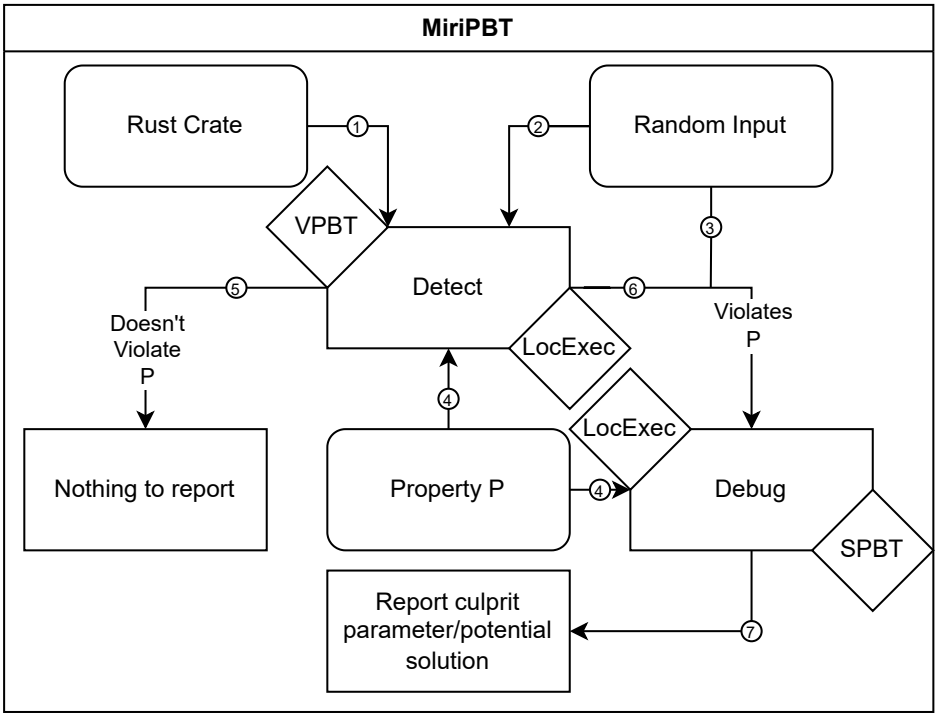

Figure 4.1: How MiriPBT works

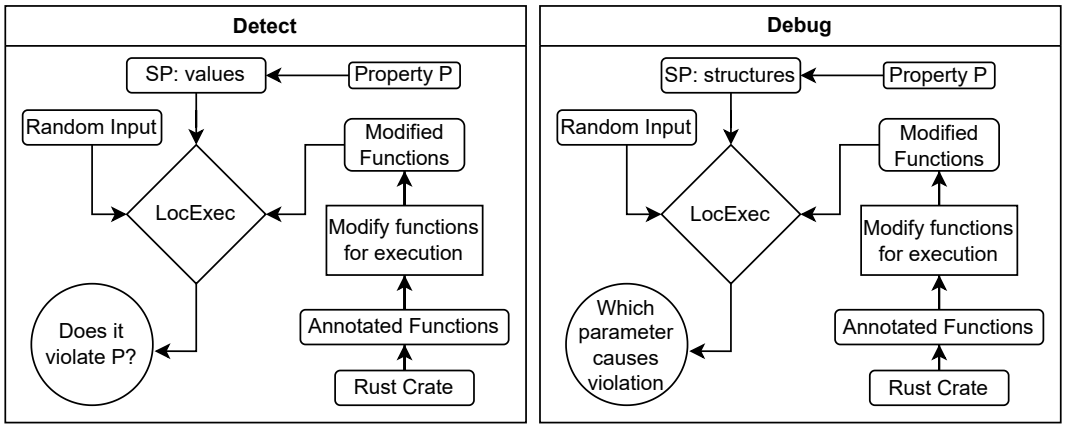

Figure 4.2: The Detect and Debug step of MiriPBT

```
fn foo(x: *const i32, y: bool) {
  if y {
    let x = x as *mut _;
    unsafe {
      *x = 42;
    }
  }
}
```

as long as the function is called with parameter `y` set to `false`, there will be no issues as far as MiriLLI is concerned. However, by using MiriPBT, we will see that setting `y` to `true` causes undefined behavior.

4.1.2 Local Execution

Local Execution is a key optimization we offer that enables MiriPBT to conduct extensive PBT based searches while still being fast. In traditional Property-Based Testing, each test input is evaluated in isolation, meaning the setup and tear down code has to be run repeatedly. These repeated executions introduce a lot of overhead, especially if a complex system needs to be set up before the test can be run.

To mitigate this problem, we introduce the technique of Local Execution (Figure 4.3). The central idea is to use process forking as a method to reuse the already initialized state across multiple tests. Specifically, once MiriPBT receives the inputs for the current round of tests from the SP, it can fork the process to execute the test in relative isolation. This fork preserves the original startup state that we can then reuse for other tests as well. This also allows us to use it as a crude benchmark, since we only focus on the hot code that our tool runs. Thus, we can heavily reduce the overhead in execution compared to traditional PBTs.

Though the idea is somewhat complex, the code behind Local Execution is relatively simple. In Algorithm 1 a simplified version of the algorithm is visible. This simplified version focuses on merely calling the fork function. It is important to note that we explicitly run each test sequentially: running them concurrently opens up the chance that one test interferes with another test. This interference can allow one test to influence another, resulting in flaky tests. By ensuring the execution happens sequentially we can make sure the tests remain deterministic and reproducible.

In summary, Local Execution is a tradeoff between efficiency and isolation: it allows more tests to run at a faster pace, as long as the outside environment does not change between test executions.

Algorithm 1 LocalExec Algorithm

Output: Either the result code of the fork or None to indicate we have forked

```
forkr ← FORK
if IS_PARENT(forkr) then
    child_status ← WAIT(forkr)
    return SOME(child_status)
else
    return NONE
end if
```

The debug stage of MiriPBT is where actual magic happens. As you can see in Figure 4.2, the debug stage looks similar to the detect stage, but instead of varying the values of the parameters, it varies the structure of the parameters. This means that it can be used to find out if there are fields that can be marked as immutable when they are normally mutable and the other way around.

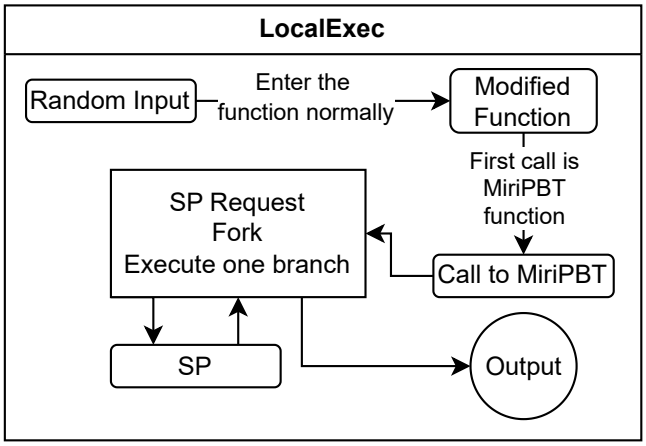

Figure 4.3: The Local Execution step of MiriPBT

This helps the developer find which fields can be marked as immutable, making sharing the parameter easier, and allowing the compiler to optimize the code better, and which fields must be mutable, which will make sure that the property is not violated in the future.

This stage is where Shrinking (Hughes 2007) is applied. We want to minimize the amount of changes required to get the program to work without violating our property. While marking every field as mutable might solve the problem, this also carries the risk of introducing new issues. Therefore, our version of shrinking starts by marking everything as mutable, and then reducing the amount of fields marked as mutable.

4.1.3 Report

MiriLLI reports only where the issue appeared and where the variable that entered the function was defined, so it often points to the function call of the offending function. This is not particularly helpful for a developer, since all it does is show where the function was used, and which parameter that the function was called with created the issue. MiriPBT's implementation, on the other hand, tries to find the definition of the struct, and reports what exact line needs to be changed, so instead of

```
Undefined Behavior: attempting a write access using <259> at alloc142[0x0],
but that tag only grants SharedReadOnly permission for this location
```

the output will look like

```
Undefined Behavior: attempting a write access for a value that only has
SharedReadOnly permissions.

Change `fn foo(x: *const i32, y: bool) -> i32' to instead use `x: *mut i32'
-> /path/to/file:line

Change `let y = &x;' to instead use `&mut x' -> /path/to/file:line

Change `let x = 10;' to instead use `let mut x' -> -> /path/to/file:line
```

Thus giving a clearer description of what can solve the problem. Do note that these advices cannot be perfect, since MiriPBT has no understanding of the entire codebase, only the one function that was tested.

4.2 Design Choices

In this section we will go over some major design choices that we made for this thesis. We start off by going into choice for PBT in MiriPBT, and the different ways in which we use Property-Based Testing. Next, we continue into how MiriPBT works, starting with a section about our modified version of MiriLLI and ending with suggestions for how a developer can implement their own executor that can hook into our code. Finally, we end with a section on the generator that the PBT uses, the basis for how it works, andwhat features are required, and we explain why this is not tightly coupled with the executor.

4.2.1 Property-Based Testing

In this thesis we use PBT in the traditional way, and in a rather unconventional way. Conventionally, PBT frameworks generate tests by varying the values of inputs within a domain, often taking extra care to explore edge cases, like the maximum number an integer can hold or an empty string. We, on the other hand, additionally employ PBT to vary the structure of our inputs, specifically focusing on the mutability of an input.

This structural variation shows the central role mutability has in Rust's ownership model. Mutability not only alters whether or not a value can be edited, but also how values interact with each other. By varying the mutability we can use the Borrow Trackers themselves as a test.

Besides this, we also allow the user to decide which property they want to test, be that Stacked Borrows or Tree Borrows. Since we still hook into MIRIs borrow tracker framework, we can adjust the specific Borrow Tracker to use quite easily.

Together the structural variation of mutability and the selection of what property to test define the input space for our testing process. Either the test passes, and there is no violation, or it fails, and there is one. If we find a violation, we can start testing what specific value causes an issue. Once we find that value we can change the mutability of the arguments to solve it. By employing Shrinking (Hughes 2007), we can even find the minimal subset of changes that allows us to have the code run without violation our property.

4.2.2 Executor

The primary executor developed in this thesis is based on MiriLLI, which provides us with a framework we can use to easily detect Borrow Tracker violations while also allowing us to still test for issues in C code via FFIs. By using MiriLLI we have been able to trace the execution between Rust and external libraries (mostly in C), ensuring that no Borrowing rules were broken at runtime. This is a crucial part of our goal, since FFIs is one of the most common ways that undefined behavior can enter a Rust program.

In addition to this MIRI-based executor, it is also possible to build an executor via a crate. This framework is based on the original input created by the SP, and allows developers and researchers to implement one with minimal overhead. This can be attractive for a number of reasons:

- Different research contexts may be interested in different parts of Rust's language. One example would be simulating embedded environments, by using an executor to intercept calls to driver functions.

- Modular design can be used to easily swap out a small part of the code that a library uses. By wrapping a single function with our marker it is now possible to alter what the function does quite quickly, without replacing the original source code at all, since the marker is already there.

- It also becomes possible to time the length of a function, since we have a call to a function before and after the main body is executed it is possible to test whether an implementation speeds up execution or slows it down.

The interface is kept quite simple: for every function that one wants to intercept, an interceptor is needed, and a general exit function needs to be implemented. Do note however that this exit function is not required to quit the program: it simply runs before the method is exited. These functions would look like this:

```
1  extern "Rust" fn miripbtexit() { }
2  extern "Rust" fn miripbt_foo(argument_names: &[&str],
3                               x_ref: &*const i32,
4                               y_ref: &bool
5              ) { }
```

These two functions can now serve as an entrypoint to the function `foo` as shown in subsection 4.1.1. One can, for example, use the generator from the next section to dynamically alter the value of the function.

This also shows the separation between the generator and the executor: the generator just returns a value when requested, while the executor decides how to insert the value into the running application. The most difficult part is that an executor needs to support a wide variety of function signatures, however, generalizing over this with a trait should be possible.

4.2.3 Generator

The generator is responsible for generating the values of the PBT arguments. In most of this thesis we refer to an implementation of the generator: the SP. As was explained before, the generator needs to both generate values and structures. However, the executor decides which of the two is requested, and if the executor only makes use of values, there is no need for the generator to support structures.

The communication between the generator and executor happens over a TCP socket. Since the executor is expected to be longer lived, it handles the server, while the generator connects to the server as a client. After this, communication between the two happens over a text-based framework. One of the crates we provide contains a framework for communicating between the generator and executor in Rust.

Once the generator is connected to the executor, it is sent a json object containing the data about all the functions and arguments that can be requested. Next, the generator will wait for a request from the executor, either containing a request for a value or a structure. The generator will then generate the requested format and send it back. After this, the executor can either request a new value the same way it requested the original value, or, more interestingly, report the result of the previous execution and request a new value. In the second case, the generator can employ something like shrinking to generate a new related value. In the SP, shrinking is only applied for the structure generator: the structures it generates try to stay as closely as possible to the original input, making sure the user has to change the smallest amount of fields.

Our primary motivation to make the generator not be a built in part of MiriPBT is that editing the generator would require a lengthy compilation process. With it being a separate (possibly very small) tool, editing the way it works and altering how it generates values becomes very easy.

The generator can be implemented in any language, as long as it follows the communication protocol. However, we do provide a Rust implementation, called the SP, that can be used out of the box.

4.3 Transformations

MiriPBT uses Rust's type system to transform a function into an automatic test. Our previous function had a signature of `fn foo(x: *const i32, flag: bool)`, Rust's compiler will transform this into a signature like this:

```
FnSig {
    ident: "foo",
    args: [
        Arg {
            ident: "x",
            value: Pointer {
                mutability: Not,
                value: Primitive::I32,
            },
        },
        Arg {
            ident: "flag",
            value: Primitive::Bool,
        },
    ],
    output: None,
}
```

This we can then transform into the format for MiriPBT:

```
{
    "foo": {
        "args": {
            "x": {
                "type": "Pointer",
                "value": {
                    "type": "Value",
                    "value": "i32",
                }
            },
            "flag": {
                "type": "Value",
                "value": "bool"
            }
        }
    }
}
```

4.3.1 Macro Transformations

Rust has a built-in system to modify the code that is in a program called macros. Using these macros we can add our marker to the source code, so the transformations can be applied to it.

For this we use two macros: one that transforms a single function, and one that allows transforming an entire `extern` block. As an example the following function

```
#[miripbt_macros::marker]
fn run(i: *const i32, flag: bool) {
```

```
    // body
}
```

will be transformed to

```
#[cfg(not(miripbt))]
#[cfg_attr(miripbt_gen, miripbt::tool)]
fn run(i: *const i32, flag: bool) {
    // body
}

#[cfg(miripbt)]
fn run(i: *const i32, flag: bool) {
    extern "Rust" {
        fn miripbt_run(names: &[&str], i: &*const i32, flag: &bool);
        fn miripbtexit();
    }
    unsafe {
        miripbt_run(&["i", "flag"], &i, &flag);
    }
    let res = {
        // body
    };
    unsafe {
        miripbtexit();
    }
    res
}
```

this transformation produces two functions, one that is used when MiriPBT is not active, and one that is used when it is. The version that is used when MiriPBT is not active is also the version that runs when the translator at the beginning of section 4.3 is running. At that time the marker `miripbt::tool` is used to tell the tool to focus on these functions for transformations. The transformation defines two different functions that MiriPBT must provide: one called `miripbt_run`, called the hook ('run' is replaced by the name of the calling function), and `miripbtexit`, used to stop the actual test from completing.

The hook is the function that is used as the entrypoint for local execution. It requires the original arguments as strings as well, since Rust has already removed the names in MIR. Additionally, it takes an immutable reference to the actual argument. These are immutable to prevent changing some inner semantics of the function, where they can potentially allow a variable to be modified. In MiriPBT the mutability requirement is temporarily disabled when we alter the contents of a field, which allows us to modify what the references point to.

An alternative setup that has been used during the evaluation was to mark an entire `extern` block, resulting in all (or only specified functions) being given an entrypoint for MiriPBT. As an example, if we have the following `extern` block

```
#[miripbt_macros::marker_extern(run)]
extern "C" {
    fn run(i: *const i32, flag: bool);
    fn other(x: i32, y: *const i32) -> i32;
}
```

It would only convert the 'run' function, since there is a list of function identifiers provided. Would this be added, it would convert the function 'other' as well. The result looks like this:

```
extern "C" {
    #[link_name = "run"]
    fn __miripbt_fn_run(i: *const i32, flag: bool);
    fn other(x: i32, y: *const i32) -> i32;
}
#[cfg(not(miripbt))]
#[cfg_attr(miripbt_gen, miripbt::tool)]
fn run(i: *const i32, flag: bool) {
    unsafe { __miripbt_fn_run(i, flag) }
}
#[cfg(miripbt)]
fn run(i: *const i32, flag: bool) {
    extern "Rust" {
        fn miripbt_run(names: &[&str], i: &*const i32, flag: &bool);
        fn miripbtexit();
    }
    unsafe {
        miripbt_run(&["i", "flag"], &i, &flag);
    }
    let res = { unsafe { __miripbt_fn_run(i, flag) } };
    // exit
    res
}
```

the FFI templates get replaced by unsafe functions that call the original extern function, now renamed to prevent conflicts. This allows us to add our own call to an extern function, while not modifying the way that it interacts with the C code.

This transformation is more useful if all functions extern blocks need to be marked, or when you want to wrap an extern function that does not have a safe wrapper available.

4.3.2 Type extraction

To perform the actual type extraction, we introduce a secondary tool named `pbtgen`. This tool is implemented as a Rusc Driver, meaning it integrates directly with the compiler itself. Unlike a procedural macro or a standalone parser, a Rustc Driver has full access to the compiler's query features. This means we can directly access the AST and HIR, the High level intermediate representation. This allows `pbtgen` to extract a large amount of data from a single query.

The extraction process works as follows: first, the tool queries the compiler for functions annotated with a marker: `#[miripbt_gen::tool]`. If it finds a function, it will request the type signature from the compiler. This signature is then transformed into the input format described in section 4.3. Once extracted, the type information is saved as JSON data. This has two goals:

- Language agnostic representation: we don't require any of the other tools to be written in Rust, so we make sure to output the data in a format that most programming languages have a library for.
- Decoupling: We explicitly want `pbtgen`, the generator and the executor to be separate programs. This allows us to replace one part while still using the other parts.

The JSON format is defined in a separate crate that `pbtgen`, MiriPBT and SP all depend on. This guarantees that the three always use the same format and that the format is always interpreted in the same way.

In summary, `pbtgen` can be seen as the builder of the bridge between MiriPBT and the SP.

Chapter 5

Evaluation

To test the validity of our research, we answer the following research questions and one use case. We have decided to focus on the effectiveness, efficiency and quality of MiriPBT. Additionally to answering these main research questions, we wanted to highlight the improved output as well. However, as that is more focused on the end users of MiriPBT, we decided not to add it as a research question, but rather as a use case.

RQ1: (effectiveness) Does MiriPBT find more violations of Stacked Borrows when compared to MiriLLI?
RQ2: (efficiency) Is it faster to use MiriPBT than to manually modify crates to find the violations of Stacked Borrows with MiriLLI?
RQ3: (quality) Can we provide better quality reports on the source of bugs found with Stacked Borrows?
Use Case: (usability) Is the output of MiriPBT more user friendly than the output of MIRI and MiriLLI?

Dataset. The dataset we use for evaluation is the same dataset as used by MiriLLI (McCormack, Sunshine, and Aldrich 2025). This dataset consists of various Rust crates that use FFI to call into C code. The dataset has already been filtered to ensure that all crates can be built with MiriLLI. In total 23318 different tests were ran, of which 2062 worked in MiriLLI. These tests are the ones we are interested in, since the tests that fail on MiriLLI because of Stacked Borrows violations will also fail on MiriPBT. These tests are in 393 different crates. Out of the 393 total crates, 56 did not compile at all, and only 30 produced the required input for our tool to run. Of these 30 crates, we focused on the 14 crates that used raw pointers in FFI communication. Information on these crates can be found in Table 5.2. From the dataset it can be seen that all of the crates we used have a big difference between the amount of lines in C and in Rust. This is mostly due to the crates simply being bindings to the C code. The crate with the smallest difference, `tweetnacly`, has this because it already includes the bindings in the source code. The rest of the crates either did not have a valid entrypoint, or did not have any tests that called into C code. Many of the crates (158) did not have any functions that could be altered by our tool, since they did not have any parameters that could be changed. 92 crates did not have an entrypoint that could be used to run the tests, This happened either due to the crate not using FFI itself and instead having a dependency that uses FFI, or due to the crate using generated bindings via a tool like Bindgen[1].

We deduplicated this dataset as well, some crates (cityhash-sys and clickhouse-rs-cityhash-sys for example) used the exact same native library with similar bindings. For this reason, we excluded one of them as long as the bindings had the same signature, since in that case the inputs from our PBT would effectively be the same.

[1] `https://docs.rs/bindgen/latest/bindgen/`

To ensure that we did not accidentally remove MiriLLIs original ability to find bugs, we have run a randomly selected subset of crates and ensured that these crates returned the same result as the original crates did.

The original dataset specified 46 tests that exposed bugs. Of these tests, 40 were unique, since some tests resulted in multiple bugs that were found. Of these 40 tests, two did not compile at all, one expected statics that did not exist and the other defined functions multiple times. One crate tried to call a function that was not available in MiriLLI, so it was excluded as well. For the remaining 37 crates, in Table 5.1 we have provided the results.

Reason	**Occurrences count**
Mutability Issues	11
Invalid Argument/Return Type	9
Memory Leak	8
Accessing Uninitialized Memory	6
Out of Bounds Memory Access	2

Table 5.1: Reasons for SB violations in the original dataset

We will go into more detail about the 11 crates with mutability issues in section 5.3. The 'Invalid Argument/Return Type' issues were related to the size of the type provided by Rust not being the same size as the type provided by C: while the specific type can be different, the size needs to be the same at all times. One example of this is that in one crate the output of the FFI call from the C side was an `uint32_t`, while on the Rust side a `u64` was expected. For the 'Memory Leak' category every time it was a Box being turned into a raw pointer, which was then simply discarded instead of dropped properly. The category 'Accessing Uninitialized Memory' was at all times caused by an attempt to access a part of memory that has been allocated to the program, but had not yet had any values inserted into it. Finally, the 'Out of Bounds Memory Access' category was related to array indices being wrong. Since C does not have a way to know just from the pointer what size an array is, it needs an extra argument specifying this. This argument was wrong in two cases, where the length sent over was wrongly interpreted by the C side resulting in an element at index `len` being accessed, instead of the final element being at `len-1`.

Since the dataset contains no other Stacked Borrows Violations, in order to further evaluate the effectiveness of MiriPBT, we have decided to introduce synthetic bugs in the original dataset. A synthetic bug is a bug that was intentionally introduced into the code, with the goal of showing that a specific tool can detect that bug. These bugs added an extra path through the function that would only be ran if the input of a function happened to trigger that path. This was done to show that not all inputs have to trigger a bug, providing a realistic application of PBT.

In total 48 of these synthetic bugs have been added. Some tests reused the same code with a different input value, so there are less synthetic bugs than there are tests. After adding these bugs, MiriLLI found issues in four tests: the three tests from `tweetnacly` and the single test from `minilzo-rs`. MiriPBT, on the other hand, found the bug in all 56 tests, showing that it does in fact have an increased search range compared to MiriLLI.

Crate	Version	C LOC	Rust LOC	Test Count
argon2-sys	0.1.0	5057	902	6
cita-snappy	0.1.5	7092	371	1
clickhouse-driver-cth	0.1.0	759	150	1
clickhouse-rs-cityhash-sys	0.1.2	736	42	1
dilithium-raw	0.1.0	30192	1592	15
farmhash-ffi	0.1.0	12181	85	2
gmod-lzma	1.0.1	51789	221	2
lzf-sys	0.1.0	1784	106	1
minilzo-rs	0.6.0	10195	291	1
pcre2-sys	0.2.6	99181	2209	1
pqcrypto-newhope	0.1.2	405450	982	8
pqcrypto-threebears	0.2.0	404895	1402	12
saxx	0.1.2	763	175	2
tweetnacly	0.1.3	1548	860	3

Table 5.2: Information on the dataset used.

Implementation The implementation of MiriPBT is done in Rust, and is available on GitHub(de Jeu 2025). The implementation consists of three parts: the main tool itself, the Rust compiler driver that allows us to extract type information from rust source code, and the structure provider. The main tool is responsible for running the tests and generating the output, while the structure provider is responsible for generating the structures used by the tests. The structure provider can be replaced by a user of MiriPBT, allowing them to use their own implementation if they want to.

5.1 RQ1: Effectiveness of MiriPBT

We compare MiriPBT to MiriLLI. Given the same dataset as the original experiments used by MiriLLI, does MiriPBT find more Stacked Borrows violations? Our primary reason for using their dataset is that we know all of these Rust crates can be built with MiriLLI.

As can be seen in section A.2, not too many crates that successfully passed MiriLLIs tests were found. Of the 30 crates that were successfully run (section A.3), only 10 tests had Undefined Behavior. Of these 10 tests, all of them were out of bounds accesses, which happened because all of the functions called had a `len: usize` parameter, which was used to determine the length of a slice. The PBT generated a value that was larger than the length of the slice, which resulted in an out of bounds access. This does not count as a violation of Stacked Borrows, since it did not have anything to do with the mutability of the parameters.

This means that our tool did not find any Stacked Borrows violations in the dataset we used. This is not surprising, since the dataset in question was the dataset also used by MiriLLI, which was already filtered to only contain crates that had no issue passing MiriLLIs test suite. As shown in chapter 3, creating a test that violates Stacked Borrows is quite easy, and while we could have ran MiriPBT on the entire dataset, we would not have found more violations than MiriLLI did.

After editing our dataset as explained at the beginning of this chapter, we have found that MiriLLI finds only the undefined behavior in the `tweetnacly` crate and the `minilzo-sys` crate, the remainder of the crates are, as far as MiriLLI is concerned, completely bug free. MiriPBT did find the expected issue of a write to a read only reference in each crate however. This proves for us that MiriPBT can find undefined behavior where MiriLLI cannot. In Table 5.3 we have compiled the result from running both MiriLLI and MiriPBT on our modified dataset.

Crate	Introduced	MiriLLI	MiriPBT
argon2-sys	1	0	6
citra-snappy	1	0	2
clickhouse-driver-cth	1	0	1
clickhouse-rs-city-hash-sys	1	0	1
dilithium-raw	6	0	4
gmod-lzma	2	0	2
lzf-sys	2	0	1
minilzo-rs	1	1	1
pcre2-sys	1	0	1
pqcrypto-newhope	5	0	1
pqcrypto-threebears	4	0	4
saxx	16	0	2
tweetnacly	2	3	3

Table 5.3: Bugs Introduced and Bugs Found by MiriLLI and MiriPBT

The `saxx` crate has this many bugs introduced since they use a macro for the definition of their functions, and only the macro was modified. Besides this, some introduced bugs were not used in any test, one such example is in `dilithium-raw`, where there are three versions of their C algorithm included, yet only two of them were tested.

5.2 RQ2: Efficiency of MiriPBT

While the computational duration of MiriPBT is expected to be higher than MiriLLI, we expect the amount of time needed by a human to make the same changes as our PBT based approach can do on its own to be a lot longer.

Considering that every test was run 10 times, and that the PBT generated a lot of different inputs, we can see that the amount of time needed to run MiriPBT is quite high. However, since the PBT generated the inputs automatically, we did not have to manually change the code to find the issues. This means that while the computational duration of MiriPBT is higher than MiriLLI, the amount of time needed by a human to make the same changes is a lot smaller. For example, the time to add the required changes to the dataset we used took around 10 minutes, while running MiriPBT took about 5 minutes per test. This time could be optimized further by running tests of different crates in parallel, which would reduce the time needed to run MiriPBT even more.

The largest speedup that MiriPBT provides is that it does not require the user to manually change the inputs to the function we want to test, all that is required is to mark the `extern` block that the function is in with a marker, or mark a specific function with a marker. This allows the user to run MiriPBT on the entire codebase, and find all issues that are related to Stacked Borrows violations, without having to manually change the code.

After manually editing our dataset, so MiriLLI and MiriPBT both have some results to show. The speedup can be seen here:

Crate	Number of Tests	Relative Execution Time
argon2-sys	6	1.01
cita-snappy	1	0.99
clickhouse-driver-cth	1	1.00
clickhouse-rs-cityhash-sys	1	0.99
dilithium-raw	15	0.97
farmhash-ffi	2	1.03
gmod-lzma	2	0.96
lzf-sys	1	1.01
minilzo-rs	1	0.99
pcre2-sys	1	1.00
pqcrypto-newhope	8	1.04
pqcrypto-threebears	12	1.29
saxx	2	1.00
tweetnacly	3	1.42

Table 5.4: Speedup of MiriPBT compared to MiriLLI, average of 5 runs per crate, lower is faster

As can be seen from our results, executing the code is almost always as fast in MiriPBT as it is in regular MiriLLI. This was expected, since we still use MiriLLI as part of our code. It is interesting, however, that the `tweetnacly` and `pqcrypto-threebears` were so much slower to execute in MiriPBT than in MiriLLI. This happened because the randomly generated input values also controlled the amount of loops the function would go through. On one of the runs, the input happened to not violate Stacked Borrows, but it did end up with a high loop value. This resulted in some outliers that were about 1 minute longer, resulting in these inflated times.

We are not concerned that the results show similar speed to MiriLLI here, since the time needed to alter the crate with our failing inputs should also be considered. While the runtime stays the same, even having to execute the crate via MiriLLI twice would double the time that it takes to get a result.

5.3 RQ3: Quality of reports of MiriPBT

We expect to at least be better at qualifying any mutability related bugs found in foreign function interfaces. To prove this we have ran MiriPBT on the 11 mutability related issues as discussed before in the dataset section.

Crate	Bug Type
bzip2	imm_and_mut
dec	imm_and_mut
dec-number-sys	write
klu-rs	write
librsync	imm_and_mut
littlefs2	write
lsmlite-rs	write_ptr
ms5837	write
sgp4-rs	write_ptr
spng	imm_and_mut
blitsort-sys	write

Table 5.5: Mutability Bugs

In Table 5.5 we have the 11 mutability issues classified as follows:

- **imm_and_mut**: This bug was caused by creating a mutable and immutable raw pointer at the same time. While this is allowed when using the Borrow Checker, Stacked Borrows specifically forbids this since the immutable pointer will be invalidated when the mutable one is made.
- **write_ptr**: This bug was caused by attempting to write to an immutable raw pointer, while the value that the pointer points to is mutable. These bugs are easily fixed by changing the mutability of the pointer itself.
- **write**: This bug is one we are interested in, it is a write to an immutable pointer from the C side of the code. While Rust itself allows this, MiriPBT can detect this and show an error when it happens.

These **write** bugs are the interesting bugs for us, as they are the ones that MiriPBT can find solutions for. MiriPBT did in fact find solutions for all these bugs, since all of them are even in the original dataset shown to be related to a wrong signature on the Rust side, and not necessarily an issue on the C side.

5.4 Use Case: Usability of MiriPBT

We expect the usability of MiriPBT to be higher than MiriLLI when using the output generated by Stacked Borrows. While MIRI is made mostly for developers that care about the smaller details of the code, we want to create methods to find undefined behavior easier to use for more inexperienced developers. The output that MiriLLI provides is highly detailed; however, it is not clear what actually caused the issue, only how we got to the issue.

As an example, MiriLLI's output for a test in the `clickhouse-driver-cth` crate looks like this:

```
---- Foreign Error Trace ----

@ store i8 97, ptr %21, align 1, !dbg !365, !tbaa !359

/root/thesis/mirilli/work/clickhouse-driver-cth-0.1.0/src/cc/city.cc:367:10
src/lib.rs:66:1: 66:33
----------------------------

error: Undefined Behavior: writing to alloc12 which is read-only
  |
  = note: writing to alloc12 which is read-only
  = note: (no span available)
  = help: this indicates a bug in the program: it performed an invalid operation,
          and caused Undefined Behavior
  = note: BACKTRACE on thread `unnamed-2`:
```

In contrast, MiriPBT means to bring the error reporting to the same level as regular Rust compiler errors, including suggestions on how to fix the problems. As such, we implemented a clear format that includes exactly which values should have their mutability changed and which should stay the same. From a usability perspective, this distinction between highly detailed output and clear error reporting with solutions is critical.

Structured output in a familiar format makes it easier for users to understand what is happening and how to fix the problem. Therefore, we expect that the error reporting MiriPBT provides will create a noticeable improvement in usability compared to MiriLLI.

We try to focus only on improving the function itself. Once the signature of the function is changed, Rust's own error reporting can take over to show the user what else has to change. The main limitation that comes with this is that it only reasons about the input parameters to the function, so if a non-`extern` function is marked, it will try to suggest changes based on the inputs to that function instead of any native bindings used. Since `clickhouse-driver-cth` uses a simple `extern` block marker, the output is quite clear:

```
1  Try marking the input of the function as mutable:
2      --> src/lib.rs:66:29
3       |
4  66   | pub fn _CityHash128(s: *mut c_char, len: usize) -> Hash128;
5       |                         ^^^
```

In conclusion, making error reporting clearer helps make solutions more accessible. This use case proves that usability helps enable developers to create safer code, considering the positive responses to Rust's regular error reporting.

5.5 Threats to Validity

In this section we will go over the threats to validity of our evaluation. We will discuss the internal and external validity of our evaluation, and how we tried to mitigate these threats.

5.5.1 Internal Validity

The dataset we used is a very small subset of the original dataset, which means that the results we got might not be representative of the entire dataset. However, since the original dataset was already filtered to only contain crates that could be built with MiriLLI and passed all tests, we expect that the results we got are representative of the entire dataset. Any tests that failed on MiriLLI would also fail on MiriPBT, since MiriPBT is built on top of MiriLLI. A solution for the small dataset would be to run MiriPBT on the entire dataset, but this would not yield any new information, since we already know that any tests that fail on MiriLLI would also fail on MiriPBT since they are built on the same engine.

The synthetic bugs we introduced into the dataset might be another threat to the internal validity. Since they are synthetic, the results we yielded might not be representative of real world bugs. However, the synthetic bugs we introduced are very similar to real world bugs, and therefore we do expect that the results we got are representative of real world bugs. The synthetic bugs we introduced were all related to Stacked Borrows violations, which is the main focus of our tool.

5.5.2 External Validity

Another threat to validity is the randomness introduced by the PBT. Since the PBT generates random inputs, it is possible that the results we got are not representative of all possible inputs. However, since we ran each test multiple times, we do expect them to be.

Finally, since MiriPBT is built on top of MiriLLI, it inherits all the limitations of MiriLLI. This means that any limitations of MiriLLI are also limitations of MiriPBT. For example, if MiriLLI cannot handle a certain type of code, then MiriPBT will also not be able to handle that type of code.

5.6 Challenges and Limitations

The largest challenge of MiriLLI currently is that the Rust version it is based on is one year old. This means that a lot of modern features are not available. Thankfully, this was only an issue for a small number of crates, since `cargo` uses the latest version available by default. This resulted in some crates not building correctly since they used an outdated way of setting `cargo` options.

It was considered infeasible to update MiriLLI to a more recent version of Rust, because the compiler has received a lot of changes in the time between when the project was last worked on and the time of writing this thesis. If a larger time window had been available, re-implementing MiriLLI itself would probably have been a better choice than trying to bring it up to date by applying patches.

Not only the version of Rust used was out of date: the version of LLVM (16) was also surprisingly problematic. LLVM keeps the only last 3 versions easily available, which would be LLVM 20, 19 and 18 at the time that this thesis started. This, combined with the fact that older LLVM versions are not guaranteed to compile on newer compilers, resulted in us needing a Ubuntu Virtual Machine that still shipped the older LLVM versions.

One limitation that we also had was the relatively small dataset. While this was a choice that we made, it also proved to be very restrictive of the variation of data that we received. In combination with the crates that did not compile, we ended up having to limit our already small dataset even more. For any future research we recommend first updating MiriLLI to a more recent version of Rust and LLVM, and then implementing a tool based on that. The time that it takes to update MiriLLI is worth it, as you gain a rather large population of crates to test from the new crates available on `https://crates.io/`.

Chapter 6

Related work

6.1 MIR Analysis

Our tool is not the first tool to do analysis using MIR. We have talked extensively about MIRI (Miri Contributors 2025) and MiriLLI (McCormack, Sunshine, and Aldrich 2025), but there are also other tools that analyze MIR.

MirChecker (Li, Wang, Sun, and J. C. Lui 2021) is a tool that does static analysis of MIR code. It works via analysis of Rust's MIR code, detecting potential crashes and memory safety issues, all while giving users clear diagnostics on what can go wrong at which point. Since it is a static analysis tool, it does not have the same limitations as our tool. It however does not support FFI calls, so it cannot check if the Rust code is following the rules of ownership and borrowing when calling into C code.

The Kani Rust Verifier (VanHattum et al. 2022) checks an entirely different part of Rust, namely dynamic trait objects (like `dyn Foo`). Dynamic trait objects are used quite often in Rust code, but verifying them is often not assumed to be necessary, since the Rust compiler should ensure that they are used correctly. While our tool uses MIR to analyze FFI calls, Kani goes the other direction and uses it to verify Rust itself.

6.2 FFI analysis

Extensive research has been done on FFI usage in Rust. The most common topic found has to do with memory safety issues, which is exactly what we help to solve.

One paper that talks about FFI usage is by the same author as MirChecker: (Li, Wang, Sun, and J. Lui 2022). While Rust is getting used more and more for security-critical sections of code, it is still haunted by the FFI boundary. They built a tool that can be used to analyze Rust code for issues with memory management issues across the Rust/C FFI boundary. While this tool can detect similar issues as ours, it does not use PBT to find solutions to the issues it finds. Its limitation of only using static analysis is something we have solved by using PBT to generate inputs for the function we want to test.

Another relevant paper talks about security issues that can arise from incorrect FFI usage (Martin et al. 2021). When Rust's `unsafe` parts are used, memory safety violations like use after free, double free and buffer overflows can occur. These issues are however undetectable for the Borrow Checker. Since for our tool it is assumed that the FFI usage is correct, barring potential mutability issues, this paper focuses on a very different part of FFI usage.

Thankfully it is not only problem finding, but solutions that are proposed as well, like FFI Isolation (Kayondo et al. 2024). Unfortunately, there are still quite a few issues with this concept. The main idea this is built on is that memory used by the Rust program itself should be isolated from memory used for FFIs. This is done by using a separate memory allocator for FFIs. This method does not solve the mutability issues that our tool solves, and instead

focuses on a different part of FFI memory safety, since it now does not affect the Rust memory safety guarantees.

Hong et al. (2024) shows that operating systems need memory that can both be aliased and mutable. This goes against the Rust memory model, which does not allow mutable aliased memory. By reducing the A&M usages to patterns, it is shown that you can work around this issue in Rust. Thus, instead of solving a problem by changing mutability like our tool does, it instead changes the way the memory is used.

MiriPBT can help automate the analysis like was done by Qin, Chen, Yu, et al. (2020). They took a different approach from us and implemented two tools that can do static analysis, while we use dynamic analysis (in the form of Stacked Borrows).

In embedded systems, code-like libraries and drivers are commonly written in C, Schuermann, Thomas, and Levy (2023) provides a method to create automatic safe encapsulation for the FFIs required to run a program. They do this by wrapping functions using hardware-based memory protection mechanisms present on embedded chips. Compared to other approaches to make embedded programming more memory safe, this provides a lightweight solution. By using a method to switch context that is more lightweight than normally used in embedded development it speeds up context switches significantly. By making the solution very lightweight, they hope to achieve a wider adoption range of memory safe languages in the embedded domain.

6.3 Rust Type Safety

One of the earliest projects that attempts to prove the Rust language is safe is Patina (Reed 2015). It proves that a pre-1.0 version of Rust is sound. It does this by combining knowledge of certain features being sound, and creating a formal semantics of the language.

Our tool heavily leverages on the fact that Rust's type system is proven to be correct and safe. Thankfully, the RustBelt project (Jung, Jourdan, et al. 2017) proves exactly this. While this paper is not directly related to our tool, since we want to verify the type system at runtime, we require that it is correct at compile time. Since RustBelt has proven that (a subset of) Rust's type system is correct, we know that our tool can use the type system to generate inputs that are valid Rust code.

Emre, Schroeder, et al. (2021) provide a method that allows translation of C or C++ code into Rust safer than what Bindgen currently does. The translations they provide seek to create safer code than the C code it is transformed from. Besides this, they provide a method that automatically removes a case of unsafety from Rust code, combined with a method that can be extended to other kinds of unsafety. Their implemented method translates raw pointers into regular references with lifetimes by analyzing where a pointer is referenced and seeing if this can be translated into a lifetime.

However, as is shown by Emre, Boyland, et al. (2023), it remains quite difficult to translate C code to Rust safely. This paper provides a series of methods that allow improving the translations currently provided, increasing the amount of pointers translated to references by 75%. To achieve this, they have extended the set of raw pointers considered for translation into references from a small subset of all pointers to all of them. A large problem is shown to be type equality: causing a single pointer to 'infect' other pointers.

Qin, Chen, Liu, et al. (2024) conduct an empirical survey of Rust memory bugs. They focus on answering three questions: what kinds of memory safety issues Rust programs have, what kind of concurrency bugs appear and how unexpected panics are caused. This first question is related to our work, since we focus on one of the main categories of bugs that appear in Rust programs: FFI issues. They did an extensive search analyzing 70 different known bugs to create static analysis tools that have shown 93 new bugs that were not noticed before.

6.4 PBT

Testing does not only happen via unit tests, but also via Property-Based Testings. The first big library to provide PBT functionality was Haskell's QuickCheck (Claessen and Hughes 2000). It provided the foundation for what we call PBT these days. QuickCheck works via simple boolean properties in Haskell, and allows testing them via a single function call. The tool has been extended throughout the years,

One unique usage of PBT would be to verify invariants set by traits, for example Rust's `Eq` trait being reflexive. The tool built by Byrnes, Takashima, and Jia (2024) does exactly that. While we do not focus on traits (since they tend to be FFI incompatible), it does show that PBT can be used to verify properties of Rust code.

Thankfully, PBT is not just used by researchers, as is shown by Goldstein et al. (2024). Large companies use PBT to test their programs, which is especially useful when working in sizable teams. Since our tool also has a practical aspect in that it allows users to find issues in their code and explains how to fix them, it is nice to see that PBT is also used in the field.

For Cyber Security PBT is also usable (Reis 2025). Instead of fuzzing the entire input space for any bugs, using PBT to check the correctness of authentication mechanisms, cryptographic APIs and session protocols they hope to improve security. This usage of PBT is not that far removed from how we use it, since we also use it to check the correctness of memory safety mechanisms.

6.5 Test generation from types

Generating tests based on types has been done before, as shown by Seidel, Vazou, and Jhala (2015). They provide a method to turn Haskell types into SMT queries which can be solved by a tool they developed. While we seem to be the first to generate tests based on Rust's type system, it has been done before in other languages.

Chapter 7

Conclusion

We have implemented a method to generate dynamic checks based on Property-Based Testing that allows us to test for violations of Stacked Borrows in C code, and also shown that generating more accessible output is possible. This was done by modifying MiriLLI to allow for local execution of functions and to allow for the editing of values in variables. This lets us focus on a smaller part of the program, and thus speed up the testing process. Finally, we also implemented a Shrinking algorithm that allows us to find the minimal change needed to make a function follow our property again. Using this, we can generate output for a user that allows minimal changes to their code to make it safe again.

To allow for more extensive testing, we also implemented a way to generate inputs for functions based on their type signatures. This makes it possible for us to test a wider range of inputs, and thus increase the chances of finding violations of our property.

The project was divided into three separate parts: MiriPBT, the main application that the user interacts with, the SP, a program that allows the generation of inputs for functions, both for values and for mutability, and finally our type extraction tool, which extracts type information from the user's code into a format that the SP can understand.

From this, we can conclude that it is possible to use Property-Based Testing to find violations of Stacked Borrows in C code, and that it is also possible to generate more accessible output for the user. Combining these two things allows us to create a tool that can help users find and fix issues in their code, thus improving the overall quality of their code.

7.1 Future Work

Various extensions to this work can be done. Firstly, if MiriLLI were to be brought up to date with the latest version of MIRI, it would allow for more crates to be tested since Rust continues to evolve. This would make it possible for the tool to be used on a wider range of projects, and thereby increase its usefulness.

Furthermore, expanding this research to other languages that use FFI to call into C code would be interesting. Languages like Python, Ruby, and Java all have ways to call into C code, and therefore could benefit from a similar tool. This would allow for a wider range of users to benefit from this research, and thus increase its impact. While some of these languages do not have a concept of mutability quite like Rust does, it is still possible to have issues with undefined behavior when calling into C code.

Finally, expanding the types of properties that can be tested would also be interesting. While this research focused on Stacked Borrows, there are many other properties that could be tested. For example testing only a subset of Stacked Borrrows, or using Tree Borrows instead. This would allow for a wider range of issues to be found, and thereby increase the usefulness of the tool.

Bibliography

Andrews, James H et al. (2008). "Random test run length and effectiveness". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, pp. 19–28.

Astrauskas, Vytautas et al. (2020). "How do programmers use unsafe rust?" In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–27.

Byrnes, Twain, Yoshiki Takashima, and Limin Jia (2024). "Automatically Enforcing Rust Trait Properties". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Cham: Springer Nature Switzerland, pp. 210–223. ISBN: 978-3-031-50521-8.

Claessen, Koen and John Hughes (2000). "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, pp. 268–279. ISBN: 1581132026. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

Emre, Mehmet, Peter Boyland, et al. (2023). "Aliasing limits on translating C to safe Rust". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1, pp. 551–579.

Emre, Mehmet, Ryan Schroeder, et al. (2021). "Translating C to safer Rust". In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA, pp. 1–29.

Evans, Ana Nora, Bradford Campbell, and Mary Lou Soffa (2020). "Is rust used safely by software developers?" In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 246–257.

Goldstein, Harrison et al. (2024). "Property-based testing in practice". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13.

Hong, Jaemin et al. (2024). "Taming shared mutable states of operating systems in Rust". In: *Science of Computer Programming* 238, p. 103152.

Hughes, John (2007). "QuickCheck Testing for Fun and Profit". In: *Practical Aspects of Declarative Languages*. Ed. by Michael Hanus. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–32. ISBN: 978-3-540-69611-7.

de Jeu, Julius (2025). *Master Thesis Files*. URL: https://github.com/J00LZ/master-thesis-files.

Jung, Ralf (2016). *The Scope of Unsafe*. URL: https://www.ralfj.de/blog/2016/01/09/the-scope-of-unsafe.html.

Jung, Ralf, Hoang-Hai Dang, et al. (2020). "Stacked borrows: an aliasing model for Rust". In: *Proceedings of the ACM on Programming Languages* 4.POPL. DOI: 10.1145/3371109. URL: https://doi.org/10.1145/3371109.

Jung, Ralf, Jacques-Henri Jourdan, et al. (2017). "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL, pp. 1–34.

Kayondo, Martin et al. (2024). "Assessing The Landscape: A Survey on Foreign Function Interface Isolation in Rust". In: *Annual Conference of KIPS*. Korea Information Processing Society, pp. 310–313.

Li, Zhuohua, Jincheng Wang, Mingshen Sun, and John Lui (2022). "Detecting cross-language memory management issues in rust". In: *European Symposium on Research in Computer Security*. Springer, pp. 680–700.

Li, Zhuohua, Jincheng Wang, Mingshen Sun, and John CS Lui (2021). "MirChecker: detecting bugs in Rust programs via static analysis". In: *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pp. 2183–2196.

Martin, Kayondo et al. (2021). "A Study on Security Issues Due to Foreign Function Interface in Rust". In: *Annual Conference of KIPS*. Korea Information Processing Society, pp. 151–154.

Matsakis, Nicholas D. and Felix S. Klock (2014). "The rust language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: Association for Computing Machinery, pp. 103–104. ISBN: 9781450332170. DOI: 10.1145/2663171.2663188. URL: https://doi.org/10.1145/2663171.2663188.

McCormack, Ian, Tomas Dougan, et al. (2024). "A mixed-methods study on the implications of unsafe Rust for interoperation, encapsulation, and tooling". In: *arXiv preprint arXiv:2404.02230*.

McCormack, Ian, Joshua Sunshine, and Jonathan Aldrich (2025). *A Study of Undefined Behavior Across Foreign Function Boundaries in Rust Libraries*. arXiv: 2404.11671 [cs.SE]. URL: https://arxiv.org/abs/2404.11671.

Miri Contributors (2025). *Miri: An interpreter for Rust's mid-level intermediate representation*. URL: https://github.com/rust-lang/miri.

Qin, Boqin, Yilun Chen, Haopeng Liu, et al. (2024). "Understanding and detecting real-world safety issues in Rust". In: *IEEE Transactions on Software Engineering* 50.6, pp. 1306–1324.

Qin, Boqin, Yilun Chen, Zeming Yu, et al. (2020). "Understanding memory and thread safety practices and issues in real-world Rust programs". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 763–779.

Reed, Eric (2015). "Patina: A formalization of the Rust programming language". In: *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* 264.

Reis, Manuel J. C. S. (2025). "Property-Based Testing for Cybersecurity: Towards Automated Validation of Security Protocols". In: *Computers* 14.5. ISSN: 2073-431X. DOI: 10.3390/computers14050179. URL: https://www.mdpi.com/2073-431X/14/5/179.

Schuermann, Leon, Arun Thomas, and Amit Levy (2023). "Encapsulated functions: fortifying rust's ffi in embedded systems". In: *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, pp. 41–48.

Seidel, Eric L., Niki Vazou, and Ranjit Jhala (2015). "Type Targeted Testing". In: *Programming Languages and Systems*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 812–836. ISBN: 978-3-662-46669-8.

Shanker, Sid (2018). *Safe Concurrency with Rust*. URL: https://squidarth.com/rc/rust/2018/06/04/rust-concurrency.html.

Sharma, Ayushi et al. (2024). *Rust for Embedded Systems: Current State, Challenges and Open Problems (Extended Report)*. arXiv: 2311.05063 [cs.CR]. URL: https://arxiv.org/abs/2311.05063.

Stack Overflow (2022). *2022 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages.

— (2023). *2023 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages.

— (2024). *2024 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages.

— (2025). *2025 Stack Overflow Developer Survey*. URL: https://survey.stackoverflow.co/2025/technology#2-programming-scripting-and-markup-languages.

The Fuchsia Developers (2024). *Unsafe code in Rust*. URL: https://fuchsia.googlesource.com/fuchsia/+/a5baa83b59a6838ce1e2d1cec8f1d003cb693934/docs/development/languages/rust/unsafe.md.

The Rust Programming Language Contributors (2025a). *References and Borrowing - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html.

— (2025b). *The Borrow Checker - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html#the-borrow-checker.

— (2025c). *The Rust Programming Language*. URL: https://github.com/rust-lang/rust.

— (2025d). *What is Ownership? - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

VanHattum, Alexa et al. (2022). "Verifying dynamic trait objects in rust". In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pp. 321–330.

Villani, Neven (2023). "Tree borrows". In: *Accessed March,. MA thesis. ENS Paris-Saclay*.

Zhu, Shuofei et al. (2022). "Learning and programming challenges of rust: A mixed-methods study". In: *Proceedings of the 44th International Conference on Software Engineering*, pp. 1269–1281.

Appendix A

Crate Results

A.1 Status explanation

In Table A.1 we show the general status of the crates we tested. It contains the status indicator we used, the amount of crates that had this status, and a short explanation of the status.

Status	Count	Reason
onlyempty	158	All functions that called into C code did not have a parameter to alter.
no_entry	92	There was no reasonable entrypoint for this crate, since the tests did not call into C code.
external_tests	47	The tests were defined in a seperate crate from the code itself, which is not supported.
success	30	The crate executed succesfully, and we have results from the crate.
semver	16	A dependency of the crate broke semantic versioning requirements and removed a required feature on a patch upgrade.
invalid	13	The crate used syntax that became invalid.
unsupported	9	The crate used unsupported syntax in extern blocks (mostly variadic arguments)
missing	8	A feature was removed from rustc, so this crate does not compile anymore.
added	7	The crate did not initially have any valid tests, but one was found that was added later. The addition however did not result in any new results.
library	3	The crate depends on a native library that was not installed, or installed with the wrong version.
compile_error	3	The crate did not want to compile at all.
required	3	A dependency could not be built.
lints	2	A defined lint in the crate itself prevents cargo from building the crate.
notfound	2	A dependency of this crate is completely removed from crates.io, not just yanked

Table A.1: Compilation status of the crates.

A.2 Compilation Results

In Table A.2 we show the results of compiling the crates we had in our dataset. The crates that were successful are then tested, with the results shown in Table A.3.

Crate	Version	Status
a-half	0.1.0	success
absperf-minilzo	0.3.4	success
absperf-minilzo-sys	2.10.0	onlyempty
adbutils	0.1.1	no_entry
aigc_secp256k1zkp	0.7.12	unsupported
alloca	0.4.0	no_entry
angrylion-rdp-plus-sys	0.2.0	library
apecrunch	0.0.3	missing
apriltag-image	0.1.0	external_tests
apriltag-nalgebra	0.1.0	no_entry
apriltag-sys	0.3.0	unsupported
arcdps-imgui	0.8.0	no_entry
archive-lp	0.2.3	semver
argon2-sys	0.1.0	success
argonautica	0.2.0	semver
ark-msm	0.3.0-alpha.1	invalid
ast-grep-tree-sitter-c-sharp	0.20.0	onlyempty
astrology	3.0.3	no_entry
aubio	0.2.1	added
aubio-rs	0.2.0	added
babble	0.2.0	invalid
bad64	0.6.0	external_tests
basis-universal	0.3.0	no_entry
bc-components	0.4.0	no_entry
bc-crypto	0.2.0	no_entry
bc-envelope	0.5.0	external_tests
bchlib	0.2.1	added
bdflib	0.4.4	no_entry
bdk_chain	0.5.0	invalid
bip322-simple	0.3.1	no_entry
bip351	0.4.0	no_entry
bip47	0.3.0	no_entry
bitbox-api	0.1.6	compile_error
bitsign	0.1.1	no_entry
blackmagic-sys	0.1.0	external_tests
bliss-audio-aubio-rs	0.2.1	added
bloomfilter-rs	0.1.0	missing
blosc-src	0.2.1	external_tests
blosc2-src	0.1.1	external_tests
brotli2	0.3.2	external_tests
bsalib	0.1.0	no_entry
btc-transaction-utils	0.9.0	no_entry
btreec	0.3.0	onlyempty
c-closures	0.3.1	semver

capstone	0.11.0	success
cashu	0.0.2	semver
cat_solver	0.1.0	onlyempty
cee-scape	0.1.6	onlyempty
cita-snappy	0.1.5	success
cityhash-sys	1.0.5	external_tests
clickhouse-driver-cth	0.1.0	success
clickhouse-driver-lz4	0.1.0	onlyempty
clickhouse-rs	1.1.0-alpha.1	no_entry
clickhouse-rs-cityhash-sys	0.1.2	success
clickhouse-srv	0.3.1	no_entry
cpc	1.9.2	added
crlibm	0.1.1	onlyempty
crowdstrike-cloudproto	0.3.1	invalid
db3-sqlparser	0.0.1	onlyempty
dcss-api	0.1.3	external_tests
dec	0.4.8	external_tests
dec-number-sys	0.0.25	external_tests
decimal	2.1.0	required
decimal_fixes_mirror	2.0.4-fix1.0.0	required
decnumber-sys	0.1.5	onlyempty
dices	0.3.0	compile_error
digibyte	0.27.3	compile_error
dilithium-raw	0.1.0	success
elements	0.23.0	external_tests
elements-miniscript	0.2.0	no_entry
entab	0.3.1	lints
ep-capstone	0.2.0	success
errno-no-std	0.1.5	invalid
ethereum-crypto	0.2.0	no_entry
ethereum-tx-sign	6.1.2	no_entry
eu4save	0.8.2	external_tests
evercrypt_tiny-sys	0.1.1	external_tests
everrs	0.2.1	onlyempty
experimental-tree-sitter-swift	0.0.3	onlyempty
farmhash-ffi	0.1.0	success
fastlz	0.1.0	added
fft4r	0.1.0	added
fil_actor_market_v10	2.0.0	no_entry
fil_actor_market_v11	2.0.0	no_entry
fil_actors_runtime_v10	2.0.0	external_tests
fil_actors_runtime_v11	2.0.0	external_tests
final_compression	1.0.0	no_entry
flate2-crc	0.1.2	success
fluidlite	0.2.1	no_entry
fluidlite-sys	0.2.1	unsupported
fvad	0.1.3	success
generate_bitcoin_paper_wallet	0.1.0	no_entry

genetic-algorithm-tsp	0.1.3	missing
ggml	0.1.1	no_entry
gmod-lzma	1.0.1	success
gmt_m1-ctrl_outer-actuators	0.1.5	no_entry
grep-pcre2	0.1.6	no_entry
grin_secp256k1zkp	0.7.12	no_entry
guilospanck-nostr-sdk	0.1.0	semver
h3o	0.4.0	missing
h3ron	0.17.0	no_entry
hacl-star	0.1.0	external_tests
harfbuzz_rs	2.0.1	no_entry
harfbuzz_rs	2.0.1	success
hashwires	0.1.0	no_entry
highwayhash	0.0.14	semver
hurl	4.0.0	invalid
iana-time-zone-haiku	0.1.2	success
icasadi	0.1.1	external_tests
icgeek_ic_call_backend	0.2.0	semver
idcurl	0.4.3	external_tests
ifcfg	0.1.2	missing
imgui	0.11.0	no_entry
interfaces	0.0.9	missing
interfaces2	0.0.5	semver
itchy	0.2.3	semver
jpl-sys	0.0.2	external_tests
just-argon2	1.2.0	success
kaspa-consensus-core	0.0.3	no_entry
kaspa-mining	0.0.3	no_entry
keypair-rs	0.1.1	no_entry
kissat	0.1.0	onlyempty
lcms2	6.0.0	no_entry
lcms2-sys	4.0.3	unsupported
lexlib	2.0.1	semver
liana	2.0.0	missing
libastro	0.1.4	no_entry
libdeflate-sys	1.19.0	onlyempty
libdeflater	1.19.0	external_tests
libpng-sys	1.1.9	onlyempty
librsync	0.2.3	no_entry
libsecp256k1	0.7.1	external_tests
libusb-src	1.26.2	unsupported
libusb1-sys	0.6.4	unsupported
libwebp	0.1.2	onlyempty
libwebp-sys	0.9.4	onlyempty
libwebp-sys2	0.1.9	unsupported
libxdiff	0.2.0	no_entry
litcoin	0.28.1	no_entry
litcoinlib	0.29.4	no_entry
lnpbp_identity	0.9.0	no_entry

logi	0.0.7	no_entry
louis	0.6.2	no_entry
louis-sys	0.6.1	no_entry
luau-ast-rs-grammar	0.0.2	onlyempty
lz4	1.24.0	external_tests
lz4-sys	1.9.4	onlyempty
lzf-sys	0.1.0	success
lzfse	0.1.0	success
matrix-appservice-rs	0.4.0	no_entry
milagro-crypto	0.1.14	no_entry
mini-monocypher	0.1.0	external_tests
minilz4	0.6.1	external_tests
minilzo-rs	0.6.0	success
minimap2	0.1.16+minimap2.2.26	no_entry
minimap2-sys	0.1.16+minimap2.2.26	unsupported
misc_utils	4.2.4	external_tests
monacoin	0.27.1-pre	no_entry
moore-hilbert	0.1.1	success
mozjpeg-sys	2.0.2	invalid
n5	0.7.6	external_tests
nakatoshi	0.2.8	no_entry
nostr-db	0.4.2	external_tests
nostr-nostd	0.2.1	semver
nostro2	0.1.4	semver
nydus-utils	0.4.3	success
openjpeg-sys	1.0.9	success
opensrv-clickhouse	0.4.1	external_tests
owl-crypto	0.1.4	no_entry
p256k1	5.4.1	no_entry
parity-db	0.4.11	external_tests
parity-secp256k1	0.7.0	onlyempty
parquet2	0.17.2	external_tests
pcre2	0.2.4	no_entry
pcre2-sys	0.2.6	success
perfect6502-sys	0.2.0	onlyempty
pexacoin	0.1.0	no_entry
picnic-bindings	0.6.0	invalid
picohttpparser-sys	1.0.0	external_tests
pkginfo	0.1.3	external_tests
pqcrypto-newhope	0.1.2	success
pqcrypto-qtesla	0.1.1	onlyempty
pqcrypto-threebears	0.2.0	success
pure_decimal	0.0.7	no_entry
quickjs_runtime	0.11.5	semver
qurs	0.2.0	external_tests
rabbitizer	1.7.9	onlyempty
rebound-sys	0.1.1	no_entry
risc0-circuit-rv32im	0.18.0	no_entry
rlz	0.2.0	no_entry

rpi_ws281x-sys	0.1.5	onlyempty
rscompress-transformation	0.2.3	success
rsofa	0.4.5	no_entry
rsworld	0.1.0	no_entry
rsworld-sys	0.1.0	no_entry
rust-strictmath	0.1.0	onlyempty
sapio-secp256k1	0.22.4	no_entry
saxx	0.1.2	success
schematools	0.15.1	semver
secp256k1	0.27.0	external_tests
secret-toolkit-crypto	0.9.0	semver
selenite	0.6.0	required
sgp4-rs	0.4.0	onlyempty
smalloca	0.1.0	onlyempty
smoosh	0.2.1	semver
smush	0.1.5	no_entry
snarkvm-algorithms	0.14.6	missing
snarkvm-circuit-types-address	0.14.6	no_entry
snarkvm-circuit-types-boolean	0.14.6	no_entry
snarkvm-circuit-types-field	0.14.6	no_entry
snarkvm-circuit-types-scalar	0.14.6	no_entry
snarkvm-console-account	0.14.6	no_entry
snarkvm-console-network	0.14.6	no_entry
snarkvm-console-program	0.14.6	no_entry
snarkvm-ledger-coinbase	0.14.6	no_entry
snarkvm-parameters	0.14.6	no_entry
snarkvm-synthesizer-coinbase	0.13.0	no_entry
snarkvm-synthesizer-program	0.14.6	no_entry
sodalite	0.4.0	no_entry
sofa-sys	2020.7.21-beta.2	external_tests
softfloat-wrapper	0.3.4	no_entry
softfloat-wrapper-riscv	0.1.0	no_entry
special-fun	0.2.0	external_tests
speex-safe	0.6.0	no_entry
speex-sys	0.4.0	no_entry
sphinx-auther	0.1.12	no_entry
spiro-sys	0.1.1	external_tests
spng-sys	0.2.0-alpha.2	success
spritz_cipher	0.1.0	external_tests
sqlite3ext-sys	0.0.1	no_entry
stacks-common	0.0.2	no_entry
tab-hash	0.3.0	external_tests
tarantool	2.0.0	notfound
tectonic_engine_bibtex	0.2.1	onlyempty
tetsy-secp256k1	0.7.0	onlyempty
tezos_crypto_rs	0.5.1	invalid
themelio-stf	0.11.13	invalid
timelib	0.3.2	external_tests

tree-sitter-apex	1.0.0	onlyempty
tree-sitter-applesoft	3.1.1	onlyempty
tree-sitter-asena	0.0.1	onlyempty
tree-sitter-asm	0.1.0	onlyempty
tree-sitter-bash	0.20.3	onlyempty
tree-sitter-bass	0.0.2	onlyempty
tree-sitter-beancount	2.1.3	onlyempty
tree-sitter-bicep	1.0.0	onlyempty
tree-sitter-bitbake	1.0.0	onlyempty
tree-sitter-c	0.20.6	onlyempty
tree-sitter-c-sharp	0.20.0	onlyempty
tree-sitter-cairo	0.0.1	onlyempty
tree-sitter-capnp	1.5.0	onlyempty
tree-sitter-ccomment	0.20.1	onlyempty
tree-sitter-clingo	0.0.12	onlyempty
tree-sitter-cmake	0.4.0	onlyempty
tree-sitter-comment	0.1.0	onlyempty
tree-sitter-commonlisp	0.3.1	onlyempty
tree-sitter-cpon	1.0.0	onlyempty
tree-sitter-cql	0.1.1	no_entry
tree-sitter-css	0.19.0	onlyempty
tree-sitter-csv	1.1.1	onlyempty
tree-sitter-cue	0.0.1	onlyempty
tree-sitter-cwscript	0.0.5	onlyempty
tree-sitter-dart	0.0.3	onlyempty
tree-sitter-deb822	0.2.1	onlyempty
tree-sitter-dockerfile	0.1.0	onlyempty
tree-sitter-dot	0.1.6	onlyempty
tree-sitter-doxygen	1.1.0	onlyempty
tree-sitter-ebnf	0.1.0	onlyempty
tree-sitter-eds	0.0.1	onlyempty
tree-sitter-elisp	1.3.0	onlyempty
tree-sitter-elixir	0.1.0	onlyempty
tree-sitter-elm	5.6.6	onlyempty
tree-sitter-elsa	1.1.0	onlyempty
tree-sitter-embedded-template	0.20.0	onlyempty
tree-sitter-erlang	0.1.0	onlyempty
tree-sitter-func	1.0.0	onlyempty
tree-sitter-gitcommit	0.3.3	onlyempty
tree-sitter-glsl	0.1.5	onlyempty
tree-sitter-go	0.20.0	onlyempty
tree-sitter-go-sum	1.0.0	onlyempty
tree-sitter-graph	0.11.0	external_tests
tree-sitter-hare	0.20.7	onlyempty
tree-sitter-hexdump	0.1.0	onlyempty
tree-sitter-hlsl	0.1.2	onlyempty
tree-sitter-ic10	0.2.0	onlyempty
tree-sitter-icelang	0.1.6	onlyempty
tree-sitter-integerbasic	1.0.2	onlyempty

tree-sitter-ispc	0.1.0	onlyempty
tree-sitter-jack	0.1.1	onlyempty
tree-sitter-java	0.20.0	onlyempty
tree-sitter-javascript	0.20.1	onlyempty
tree-sitter-jslt	0.1.1	onlyempty
tree-sitter-json	0.19.0	onlyempty
tree-sitter-kconfig	1.0.0	onlyempty
tree-sitter-kdl	1.1.0	onlyempty
tree-sitter-kind	0.0.1	onlyempty
tree-sitter-kotlin	0.2.11	onlyempty
tree-sitter-lua	0.0.19	onlyempty
tree-sitter-luadoc	1.0.1	onlyempty
tree-sitter-luap	1.0.0	onlyempty
tree-sitter-luau	1.0.0	onlyempty
tree-sitter-lura	0.1.19	onlyempty
tree-sitter-md	0.1.5	no_entry
tree-sitter-merlin6502	2.0.0	onlyempty
tree-sitter-mozcpp	0.20.2	onlyempty
tree-sitter-mozjs	0.20.1	onlyempty
tree-sitter-nasl	0.1.0	onlyempty
tree-sitter-nickel	0.1.0	onlyempty
tree-sitter-nix	0.0.1	onlyempty
tree-sitter-noir	0.0.1	onlyempty
tree-sitter-nqc	1.0.0	onlyempty
tree-sitter-nwscript	8193.34.0-alpha.1	onlyempty
tree-sitter-objc	2.1.0	onlyempty
tree-sitter-odin	1.0.0	onlyempty
tree-sitter-onotone	0.1.1	onlyempty
tree-sitter-openscad	0.4.2	onlyempty
tree-sitter-org	1.3.3	onlyempty
tree-sitter-palm	0.1.0	onlyempty
tree-sitter-plymouth-script	0.1.0	onlyempty
tree-sitter-po	0.0.1	onlyempty
tree-sitter-pony	1.0.0	onlyempty
tree-sitter-poweron	1.0.12	onlyempty
tree-sitter-preproc	0.20.1	onlyempty
tree-sitter-prisma	0.1.1	onlyempty
tree-sitter-prisma-io	1.4.0	onlyempty
tree-sitter-puppet	1.2.0	onlyempty
tree-sitter-python	0.20.4	onlyempty
tree-sitter-qmldir	0.0.1	onlyempty
tree-sitter-qmljs	0.1.2	onlyempty
tree-sitter-query	0.1.0	onlyempty
tree-sitter-r	0.19.5	onlyempty
tree-sitter-regex	0.20.0	onlyempty
tree-sitter-ron	0.1.0	onlyempty
tree-sitter-rslox	0.1.1	onlyempty
tree-sitter-rst	0.1.0	onlyempty
tree-sitter-ruby	0.20.0	onlyempty

tree-sitter-rush	0.1.0	onlyempty
tree-sitter-rust	0.20.4	onlyempty
tree-sitter-scala	0.20.2	onlyempty
tree-sitter-sdml	0.2.0	onlyempty
tree-sitter-slint	0.1.0	onlyempty
tree-sitter-smali	1.0.0	onlyempty
tree-sitter-smithy	0.0.1	onlyempty
tree-sitter-solidity	0.0.3	onlyempty
tree-sitter-souffle	0.4.0	onlyempty
tree-sitter-sourcepawn	0.6.0	onlyempty
tree-sitter-sql	0.0.2	onlyempty
tree-sitter-sql-bigquery	0.5.0	onlyempty
tree-sitter-squirrel	1.0.0	onlyempty
tree-sitter-ssh-client-config	2023.9.14	onlyempty
tree-sitter-starlark	1.1.0	onlyempty
tree-sitter-strings	0.1.0	onlyempty
tree-sitter-swift	0.3.6	onlyempty
tree-sitter-tablegen	0.0.1	onlyempty
tree-sitter-thrift	0.5.0	onlyempty
tree-sitter-toml	0.20.0	onlyempty
tree-sitter-tsq	0.19.0	onlyempty
tree-sitter-turbowave	1.7.1	onlyempty
tree-sitter-ungrammar	0.0.1	onlyempty
tree-sitter-uxntal	1.0.0	onlyempty
tree-sitter-wdl-1	0.1.10	onlyempty
tree-sitter-wenyan	0.1.0	onlyempty
tree-sitter-wgsl	0.0.6	onlyempty
tree-sitter-whitespace	0.0.1	onlyempty
tree-sitter-yabo	0.0.1	onlyempty
tree-sitter-yuck	0.0.2	onlyempty
tree_sitter_grep_tree-sitter-rust	0.20.3-dev.0	onlyempty
trezor-crypto-lib	0.1.1	external_tests
tweetnacl	0.4.0	success
tweetnacly	0.1.3	success
twenty-first	0.32.1	no_entry
uma	0.1.0	invalid
unrar_sys	0.3.0	onlyempty
v-clickhouse-rs	0.2.0-alpha.7	no_entry
vapkey-cli	0.1.0	no_entry
webm-sys	1.0.3	onlyempty
webp-animation	0.8.0	invalid
webrtc-vad	0.4.0	onlyempty
wedpr_l_crypto_signature_secp256k1	1.1.0	invalid
wgctrl-rs	0.1.0	unsupported
wmm	0.2.3	onlyempty
xdelta3	0.1.5	external_tests
xs233	0.3.0	no_entry

xxhash-c	0.8.2	lints
xxhash-c-sys	0.8.4	external_tests
xxhrs	2.0.0	notfound
xyz	0.2.1	external_tests
xz2	0.1.7	external_tests
yatima-core	0.1.1	no_entry
ytnef	0.2.0	no_entry
zeronet_cryptography	0.1.9	no_entry
zmq	0.10.0	library
zmq2	0.5.0	library

Table A.2: Test results of the crates.

A.3 Successful Runs

In Table A.3 we show the results of the crates that returned a successful status. Every crate was ran 10 times with random inputs, and the results were collected. The table shows the crate name, the version of the crate, the test that was ran, how many times it was successful, how many times it resulted in undefined behavior, and how many times it returned other results (e.g. panics or assert macro failures).

Crate	**Version**	**Test**	**Success**	**UB**	**Other**
absperf-minilzo	0.3.4	checksum::tests::adler32	10	0	0
a-half	0.1.0	tests::arith_ops	10	0	0
a-half	0.1.0	tests::float_to_half	10	0	0
a-half	0.1.0	tests::half_to_float	10	0	0
a-half	0.1.0	tests::partial_eq_check	10	0	0
a-half	0.1.0	tests::partial_ord_check	10	0	0
a-half	0.1.0	tests::predicates	10	0	0
argon2-sys	0.1.0	argon2i_v10::case_11	10	0	0
argon2-sys	0.1.0	argon2i_v10::case_9	10	0	0
argon2-sys	0.1.0	argon2i_v13::case_22	10	0	0
argon2-sys	0.1.0	argon2i_v13::case_23	10	0	0
argon2-sys	0.1.0	argon2i_v13::case_24	10	0	0
capstone	0.11.0	test::test_arm64_none	10	0	0
cita-snappy	0.1.5	tests::invalid	10	0	0
clickhouse-driver-cth	0.1.0	test::test_city_hash_128	0	10	0
clickhouse-rs-cityhash-sys	0.1.2	test_city_hash_128	0	10	0
dilithium-raw	0.1.0	dilithium2::tests::test_deterministic_keygen	10	0	0
dilithium-raw	0.1.0	dilithium2::tests::test_invalid_modified_signature	10	0	0
dilithium-raw	0.1.0	dilithium2::tests::test_invalid_other_signature	10	0	0

dilithium-raw	0.1.0	dilithium2::tests::test_sign_verify	10	0	0
dilithium-raw	0.1.0	dilithium3::tests::test_deterministic_keygen	10	0	0
dilithium-raw	0.1.0	dilithium3::tests::test_invalid_modified_signature	10	0	0
dilithium-raw	0.1.0	dilithium3::tests::test_invalid_other_signature	10	0	0
dilithium-raw	0.1.0	dilithium3::tests::test_sign_verify	10	0	0
dilithium-raw	0.1.0	dilithium5::tests::test_deterministic_keygen	10	0	0
dilithium-raw	0.1.0	dilithium5::tests::test_invalid_modified_signature	10	0	0
dilithium-raw	0.1.0	dilithium5::tests::test_invalid_other_signature	10	0	0
dilithium-raw	0.1.0	dilithium5::tests::test_sign_verify	10	0	0
dilithium-raw	0.1.0	ffi::dilithium2::clean::tests::test_sign_verify	10	0	0
dilithium-raw	0.1.0	ffi::dilithium3::clean::tests::test_sign_verify	10	0	0
dilithium-raw	0.1.0	ffi::dilithium5::clean::tests::test_sign_verify	10	0	0
ep-capstone	0.2.0	test::test_support	10	0	0
ep-capstone	0.2.0	test::test_version	10	0	0
farmhash-ffi	0.1.0	tests::no_seed_64	0	10	0
farmhash-ffi	0.1.0	tests::seed_64	0	10	0
flate2-crc	0.1.2	tests::prop	10	0	0
fvad	0.1.3	test::is_voice_frame	10	0	0
fvad	0.1.3	test::set_mode	10	0	0
fvad	0.1.3	test::set_sample_rate	10	0	0
gmod-lzma	1.0.1	test_compress	10	0	0
gmod-lzma	1.0.1	test_decompress	10	0	0
harfbuzz_rs	2.0.1	common::tests::test_tag_creation	10	0	0
iana-time-zone-haiku	0.1.2	tests::test_fallback_on_non_haiku_platforms	10	0	0
just-argon2	1.2.0	test::test_variant_names	10	0	0
lzfse	0.1.0	tests::encode_buffer_to_small	10	0	0
lzf-sys	0.1.0	tests::roundtrip	0	10	0
minilzo-rs	0.6.0	tests::test_adler32	0	10	0

moore-hilbert	0.1.1	tests::test_increment_coordinates_beyond_bit_range	0	0	10
moore-hilbert	0.1.1	tests::test_iterate_dimensions	0	0	10
nydus-utils	0.4.3	compress::tests::test_lz4_compress_decompress_4097_bytes	7	0	3
openjpeg-sys	1.0.9	poke	10	0	0
pcre2-sys	0.2.6	tests::itworks	10	0	0
pqcrypto-newhope	0.1.2	ffi::test_newhope1024cca_clean::test_ffi	10	0	0
pqcrypto-newhope	0.1.2	ffi::test_newhope1024cpa_clean::test_ffi	10	0	0
pqcrypto-newhope	0.1.2	ffi::test_newhope512cca_clean::test_ffi	10	0	0
pqcrypto-newhope	0.1.2	ffi::test_newhope512cpa_clean::test_ffi	10	0	0
pqcrypto-newhope	0.1.2	newhope1024cca::test::test_kem	10	0	0
pqcrypto-newhope	0.1.2	newhope1024cpa::test::test_kem	10	0	0
pqcrypto-newhope	0.1.2	newhope512cca::test::test_kem	10	0	0
pqcrypto-newhope	0.1.2	newhope512cpa::test::test_kem	10	0	0
pqcrypto-threebears	0.2.0	babybearephem::test::test_kem	10	0	0
pqcrypto-threebears	0.2.0	babybear::test::test_kem	10	0	0
pqcrypto-threebears	0.2.0	ffi::test_babybear_clean::test_ffi	10	0	0
pqcrypto-threebears	0.2.0	ffi::test_babybearephem_clean::test_ffi	10	0	0
pqcrypto-threebears	0.2.0	ffi::test_mamabear_clean::test_ffi	10	0	0
pqcrypto-threebears	0.2.0	ffi::test_mamabearephem_clean::test_ffi	10	0	0
pqcrypto-threebears	0.2.0	ffi::test_papabear_clean::test_ffi	10	0	0
pqcrypto-threebears	0.2.0	ffi::test_papabearephem_clean::test_ffi	10	0	0

pqcrypto-threebears	0.2.0	mamabearephem::test::test_kem	10	0	0
pqcrypto-threebears	0.2.0	mamabear::test::test_kem	10	0	0
pqcrypto-threebears	0.2.0	papabearephem::test::test_kem	10	0	0
pqcrypto-threebears	0.2.0	papabear::test::test_kem	10	0	0
rscompress-transformation	0.2.3	bwt::tests::test_easy_roundtrip	10	0	0
saxx	0.1.2	tests::esa_from_slice	9	1	0
saxx	0.1.2	tests::esaxx_compact	7	3	0
spng-sys	0.2.0-alpha.2	create_context	10	0	0
tweetnacl	0.4.0	hashblocks_sha512_twice_eq	10	0	0
tweetnacly	0.1.3	tweetnacl::crypto_box	0	10	0
tweetnacly	0.1.3	tweetnacl::crypto_sign	10	0	0
tweetnacly	0.1.3	tweetnacl::sha512_basic	0	10	0

Table A.3: Test results of the crates.