Rust Verification using Rustviz and Kani

Kavita - u1418910@utah.edu
Sakshi Singh - u1418696@utah.edu

May 11, 2023

1 Abstract

Rust is a programming language that provides strong memory and type safety guarantees. It does not have a garbage collector to improve the performance of the programs and provide high-level abstractions. It is also used to write highly concurrent programs. Some major challenges in verifying the rust language are unsafe codes, closures, and stdlib. Some tools cannot handle unsafe codes since they assume the type of safety guarantee. Others might fail at closures because of their higher-order behavior and at stdlib for two reasons: unsafe codes and high optimization.

The objective of our project is to gain familiarity with the Rust programming language, including its distinctive characteristics such as ownership and borrowing. We will learn how to write efficient Rust code and explore the Rustviz tool to comprehend how it can be utilized to visualize the execution of Rust code. Additionally, we will utilize the Kani tool to conduct Rust verification.

RustViz is a tool designed to help users better understand Rust's ownership and borrowing mechanisms by generating interactive visualizations from simple Rust programs, whereas Kani is a bit-precise model checker for Rust, particularly useful for verifying unsafe code blocks where the compiler's "unsafe superpowers" are unchecked.

So our journey is started from rust language , we got familiar with rust and then explored rustviz . Rustviz is a visualization tool , so for verification purpose we shifted to Kani.

2 About Rust

Rust is a blazing-fast, memory-efficient, and type-safe programming language with a rich ownership model, which allows developers to write performance-critical code while guaranteeing memory and thread safety. It is designed to provide low-level control and high-level ergonomics, making it an attractive option for both systems programming and general-purpose development.

2.1 Installation and Environment Setup For Rust

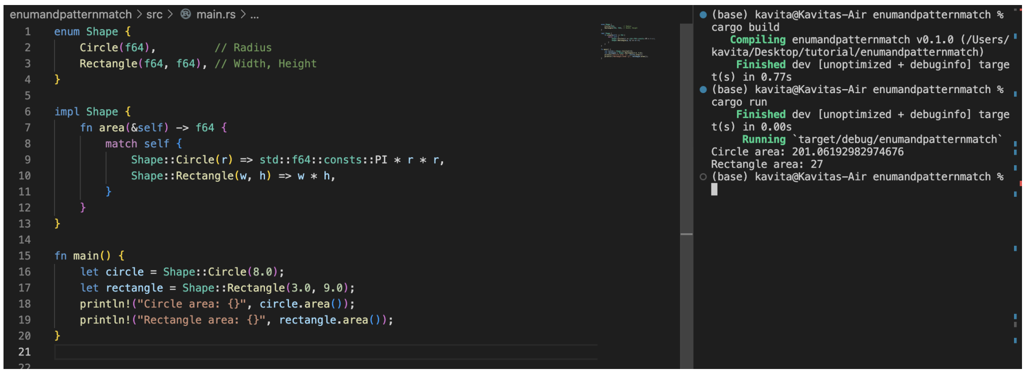
1. run the following in your terminal
   "curl –proto '=https' ––tlsv1.2 -sSf https://sh.rustup.rs|sh" (In the Rust development environment, all tools are installed to the /.cargo/bin directory, and this is where you will find the Rust toolchain, including rustc, cargo, and rustup.)
2. Download and install Visual Studio Code (VS Code) as your code editor.
3. Open a terminal and navigate to the directory where you want to create your project.
4. Create a new project using the following command: cargo new $< project\_name >$. This will create a new directory containing your project with a basic file structure.
5. Navigate to your project directory using the cd command and run cargo build. This will compile the project and create a binary executable in the target/debug directory.
6. Once the build is successful, you can run your project using cargo run. This command will both compile and run your project.

2.2 Key Features of Rust Language

1. **Type System**: Rust has a robust, static type system that prevents common programming errors such as null pointer dereferences, buffer overflows, and data races.

Type	Contents
Primitive Types	Boolean, Numeric, Textual, Never
Sequence Types	Tuple, Array, Slice
User-defined Types	Struct, Enum, Union
Function Types	Functions, Closures
Pointer Types	References, Raw pointers, Function pointers
Trait Types	Trait objects, Impl trait

Below is an example of the type system



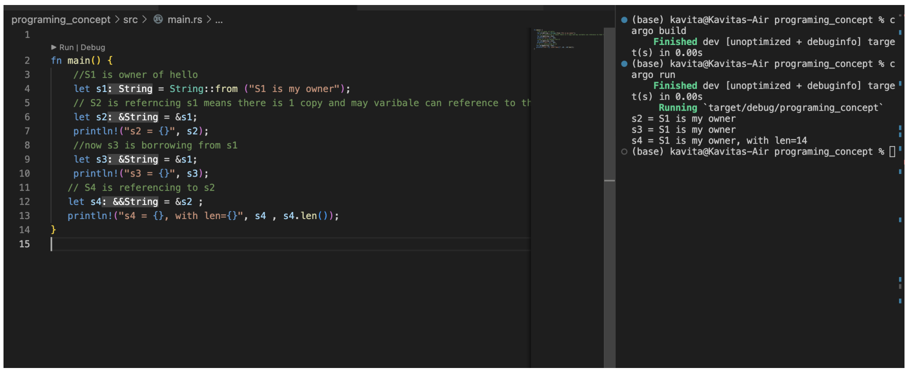
```
enumandpatternmatch > src > main.rs > ...
enum Shape {
    Circle(f64),         // Radius
    Rectangle(f64, f64), // Width, Height
}

impl Shape {
    fn area(&self) -> f64 {
        match self {
            Shape::Circle(r) => std::f64::consts::PI * r * r,
            Shape::Rectangle(w, h) => w * h,
        }
    }
}

fn main() {
    let circle = Shape::Circle(8.0);
    let rectangle = Shape::Rectangle(3.0, 9.0);
    println!("Circle area: {}", circle.area());
    println!("Rectangle area: {}", rectangle.area());
}
```

```
(base) kavita@Kavitas-Air enumandpatternmatch %
cargo build
   Compiling enumandpatternmatch v0.1.0 (/Users/
kavita/Desktop/tutorial/enumandpatternmatch)
    Finished dev [unoptimized + debuginfo] targe
t(s) in 0.77s
(base) kavita@Kavitas-Air enumandpatternmatch %
cargo run
    Finished dev [unoptimized + debuginfo] targe
t(s) in 0.00s
     Running `target/debug/enumandpatternmatch`
Circle area: 201.06192982974676
Rectangle area: 27
(base) kavita@Kavitas-Air enumandpatternmatch %
```

The provided code defines an enum called Shape which represents different shapes, such as circles and rectangles. A method called area is implemented for the Shape enum, which calculates the area of a shape based on its type using pattern matching. For instance, if the Shape is a Circle, the area method uses the radius of the circle to calculate its area using the formula $PI * r^2$. If the Shape is a Rectangle, the area method uses its width and height to calculate its area using the formula w * h. By defining a function that operates on the Shape enum, the Rust compiler can ensure that the area function is only called with a Shape value, preventing programming errors at compile time and improving the overall correctness of the program. This demonstrates Rust's support for algebraic data types and polymorphism through enums, allowing for implementing methods that operate on different shape variants.

2. **Ownership Model**: Rust's ownership system ensures that memory is used safely and efficiently without needing a garbage collector. This enables developers to reason about their code more effectively and avoid common memory-related bugs.

Example of ownership



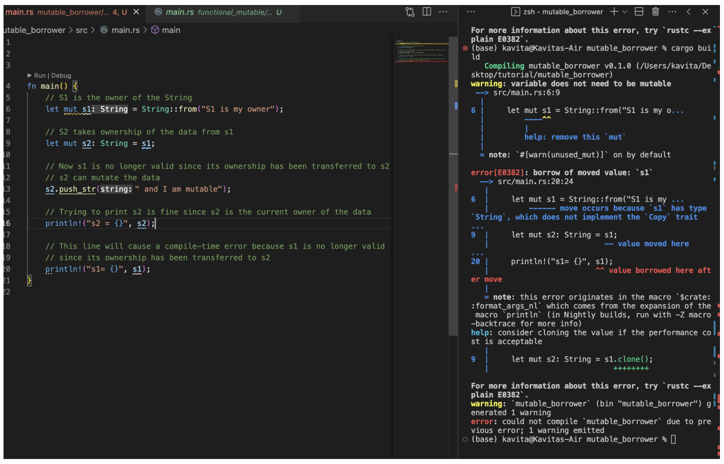
```
programing_concept > src > main.rs > ...

▶ Run | Debug
fn main() {
    //S1 is owner of hello
    let s1: String = String::from ("S1 is my owner");
    // S2 is referncing s1 means there is 1 copy and may varibale can reference to th
    let s2: &String = &s1;
    println!("s2 = {}", s2);
    //now s3 is borrowing from s1
    let s3: &String = &s1;
    println!("s3 = {}", s3);
   // S4 is referencing to s2
   let s4: &&String = &s2 ;
   println!("s4 = {}, with len={}", s4 , s4.len());
}
```

```
(base) kavita@Kavitas-Air programing_concept % c
argo build
    Finished dev [unoptimized + debuginfo] targe
t(s) in 0.00s
(base) kavita@Kavitas-Air programing_concept % c
argo run
    Finished dev [unoptimized + debuginfo] targe
t(s) in 0.00s
     Running `target/debug/programing_concept`
s2 = S1 is my owner
s3 = S1 is my owner
s4 = S1 is my owner, with len=14
(base) kavita@Kavitas-Air programing_concept %
```

The ownership model in Rust is a core concept that enables memory safety by ensuring that each value in Rust has a unique variable that is called its owner. In the above program, the ownership of the string

”S1 is my owner” is assigned to the variable s1. When s2 is created to reference s1, it points to the same memory location as s1 but does not own it. Similarly, s3 is also a reference to s1. Since s1 is the owner of the string, it can modify and release the memory when it is no longer required. When s4 is created as a reference to s2, it also points to the memory location of s1. This illustrates the concept of borrowing in Rust, where multiple variables can reference the same memory location. However, Rust’s ownership model ensures that there can only be one owner of a piece of data at a time, and this prevents data races and memory leaks by freeing memory when it is no longer needed.

Example of single ownership

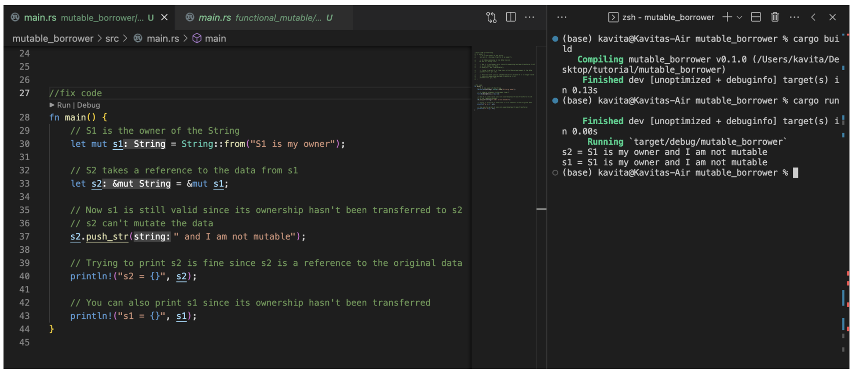

main.rs mutable_borrower/... 4, U
main.rs functional_mutable/... U
utable_borrower > src > main.rs > main
zsh - mutable_borrower

```rust
▶ Run | Debug
fn main() {
    // S1 is the owner of the String
    let mut s1: String = String::from("S1 is my owner");

    // S2 takes ownership of the data from s1
    let mut s2: String = s1;

    // Now s1 is no longer valid since its ownership has been transferred to s2
    // s2 can mutate the data
    s2.push_str(string: " and I am mutable");

    // Trying to print s2 is fine since s2 is the current owner of the data
    println!("s2 = {}", s2);

    // This line will cause a compile-time error because s1 is no longer valid
    // since its ownership has been transferred to s2
    println!("s1= {}", s1);
}
```

```
For more information about this error, try `rustc --ex
plain E0382`.
(base) kavita@Kavitas-Air mutable_borrower % cargo bui
ld
   Compiling mutable_borrower v0.1.0 (/Users/kavita/De
sktop/tutorial/mutable_borrower)
warning: variable does not need to be mutable
 --> src/main.rs:6:9
  |
6 |     let mut s1 = String::from("S1 is my o...
  |         ----^^
  |         |
  |         help: remove this `mut`
  |
  = note: `#[warn(unused_mut)]` on by default

error[E0382]: borrow of moved value: `s1`
  --> src/main.rs:20:24
   |
6  |     let mut s1 = String::from("S1 is my ...
   |         ------ move occurs because `s1` has type
`String`, which does not implement the `Copy` trait
...
9  |     let mut s2: String = s1;
   |                          -- value moved here
...
20 |     println!("s1= {}", s1);
   |                        ^^ value borrowed here aft
er move
   |
   = note: this error originates in the macro `$crate:
:format_args_nl` which comes from the expansion of the
 macro `println` (in Nightly builds, run with -Z macro
-backtrace for more info)
help: consider cloning the value if the performance co
st is acceptable
   |
9  |     let mut s2: String = s1.clone();
   |                            ++++++++

For more information about this error, try `rustc --ex
plain E0382`.
warning: `mutable_borrower` (bin "mutable_borrower") g
enerated 1 warning
error: could not compile `mutable_borrower` due to pre
vious error; 1 warning emitted
(base) kavita@Kavitas-Air mutable_borrower %
```

The code presents an issue related to ownership and borrowing in Rust. Here, the program starts by creating a String named s1, which owns the allocated memory for the string data. Then, the code transfers ownership of the String data from s1 to s2 using the let mut s2: String = s1; statement. Currently, s1 is no longer valid and cannot be used. However, the program attempts to print the value of s1, which is no longer the data owner. This causes a compile-time error because Rust ensures that only one owner exists at any given time to prevent data races and memory leaks. To fix the issue, the program must either clone the data or use references to allow multiple variables to borrow the same data without transferring ownership.

Fixing the above issue by using a reference


main.rs mutable_borrower/... U
main.rs functional_mutable/... U
mutable_borrower > src > main.rs > main
zsh - mutable_borrower

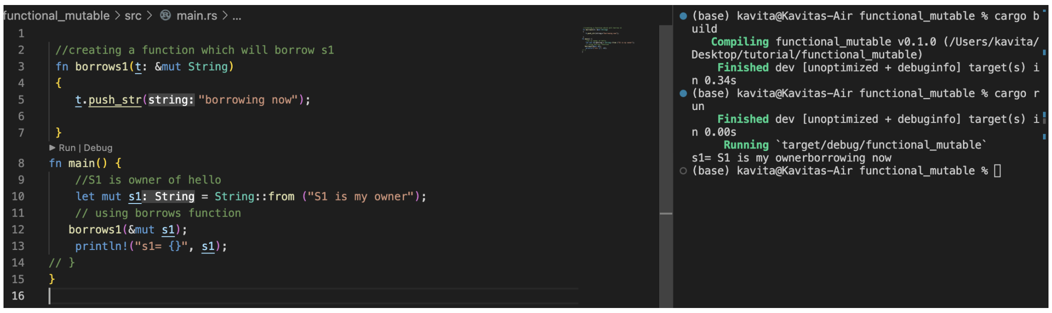
```rust
//fix code
▶ Run | Debug
fn main() {
    // S1 is the owner of the String
    let mut s1: String = String::from("S1 is my owner");

    // S2 takes a reference to the data from s1
    let s2: &mut String = &mut s1;

    // Now s1 is still valid since its ownership hasn't been transferred to s2
    // s2 can't mutate the data
    s2.push_str(string: " and I am not mutable");

    // Trying to print s2 is fine since s2 is a reference to the original data
    println!("s2 = {}", s2);

    // You can also print s1 since its ownership hasn't been transferred
    println!("s1 = {}", s1);
}
```

```
(base) kavita@Kavitas-Air mutable_borrower % cargo bui
ld
   Compiling mutable_borrower v0.1.0 (/Users/kavita/De
sktop/tutorial/mutable_borrower)
    Finished dev [unoptimized + debuginfo] target(s) i
n 0.13s
(base) kavita@Kavitas-Air mutable_borrower % cargo run

    Finished dev [unoptimized + debuginfo] target(s) i
n 0.00s
     Running `target/debug/mutable_borrower`
s2 = S1 is my owner and I am not mutable
s1 = S1 is my owner and I am not mutable
(base) kavita@Kavitas-Air mutable_borrower %
```

The given code now references s1 instead of transferring ownership to s2. This means that s1 remains the data owner, and s2 is just a temporary borrow of that data. Since s2 is a mutable reference, it can modify the data in s1, but it cannot take ownership of it. Using references, Rust allows for safe and efficient data sharing between variables and functions without the risk of ownership-related errors.

Function borrower

Here is example of mutable borrowing by using function . The below code demonstrates the concept of mutable borrowing in Rust.

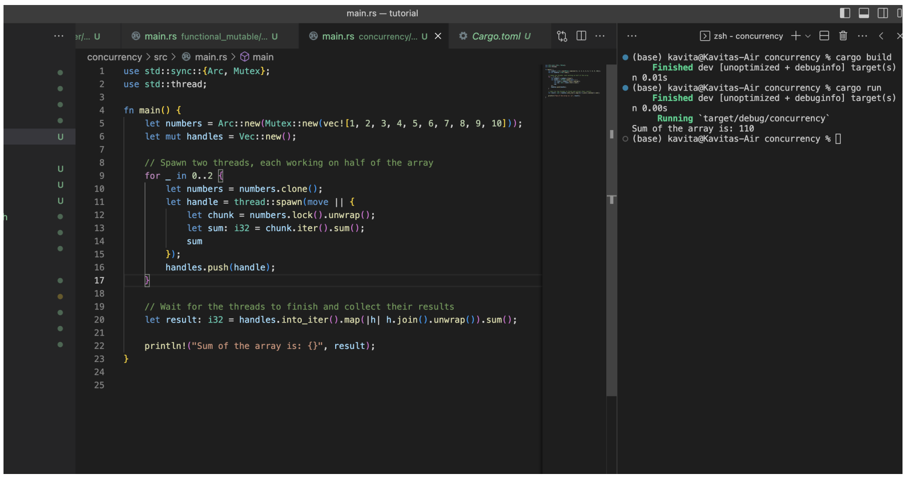


```
functional_mutable > src > main.rs > ...

//creating a function which will borrow s1
fn borrows1(t: &mut String)
{
    t.push_str(string: "borrowing now");

}
▶ Run | Debug
fn main() {
    //S1 is owner of hello
    let mut s1: String = String::from ("S1 is my owner");
    // using borrows function
   borrows1(&mut s1);
    println!("s1= {}", s1);
// }
}
```

```
(base) kavita@Kavitas-Air functional_mutable % cargo b
uild
   Compiling functional_mutable v0.1.0 (/Users/kavita/
Desktop/tutorial/functional_mutable)
    Finished dev [unoptimized + debuginfo] target(s) i
n 0.34s
(base) kavita@Kavitas-Air functional_mutable % cargo r
un
    Finished dev [unoptimized + debuginfo] target(s) i
n 0.00s
     Running `target/debug/functional_mutable`
s1= S1 is my ownerborrowing now
(base) kavita@Kavitas-Air functional_mutable % 
```

Here borrows1 function is created that takes a mutable reference to a String as a parameter, it allows the function to modify the string without taking ownership of the data. In the main function, s1 is a mutable String, and when calling borrows1, a mutable reference to s1 is passed. This approach ensures that the ownership of s1 remains within the main function, while borrows1 can still modify its content. This demonstrates Rust's ability to safely and efficiently manage memory by enforcing clear rules and restrictions around borrowing and ownership.

3. **Concurrency**: Rust supports concurrency and parallelism, making it easier to write safe and efficient concurrent code.



```
concurrency > src > main.rs > main
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let numbers = Arc::new(Mutex::new(vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10]));
    let mut handles = Vec::new();

    // Spawn two threads, each working on half of the array
    for _ in 0..2 {
        let numbers = numbers.clone();
        let handle = thread::spawn(move || {
            let chunk = numbers.lock().unwrap();
            let sum: i32 = chunk.iter().sum();
            sum
        });
        handles.push(handle);
    }

    // Wait for the threads to finish and collect their results
    let result: i32 = handles.into_iter().map(|h| h.join().unwrap()).sum();

    println!("Sum of the array is: {}", result);
}
```

```
(base) kavita@Kavitas-Air concurrency % cargo build
    Finished dev [unoptimized + debuginfo] target(s)
n 0.01s
(base) kavita@Kavitas-Air concurrency % cargo run
    Finished dev [unoptimized + debuginfo] target(s)
n 0.00s
     Running `target/debug/concurrency`
Sum of the array is: 110
(base) kavita@Kavitas-Air concurrency % 
```

The program first creates shared ownership of the vector of numbers using the Arc type, which allows multiple threads to access and modify the data safely. A Mutex locks the vector before reading and writing to ensure that only one thread can modify the vector at a time. The program then spawns two threads, each working on a different half of the vector. The threads use the Arc and Mutex types to access and modify the shared data safely. Finally, the program waits for the threads to finish and collects their results to compute the sum of all numbers in the vector. Using Rust's concurrency primitives, developers can write safe and efficient concurrent code without worrying about race conditions and other common concurrency issues.

4. **Cargo**: Rust's package manager and build tool, Cargo, simplifies dependency management and ensures consistent compilation across the Rust ecosystem. Below is an example of how to create a rust project using cargo.
   a. **cargo new project** $->$ The cargo new command creates a new Rust project with a default directory structure and a Cargo.toml file that specifies the project's dependencies and builds configuration.
   b. **cargo build** $->$ Run cargo build after navigating to the project directory. The cargo build command compiles the project and creates a binary executable in the target/debug directory,
   c. **cargo run**$->$ cargo run both compiles and runs the project.

Using Rust language, we have created many programs (binary search, bubble sort, linked list example, dequeue, guessing game, etc.). Please refer Git for the code access.

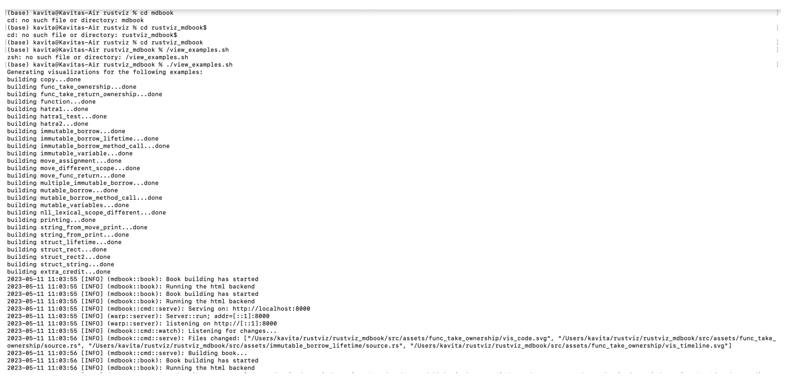
3 Rustviz (visualization tool)

Rustviz is an open-source project that provides a visualization of the execution of Rust code. It enables us to observe how information is moved between functions and variables and the control flow of our code. To analyze and debug the code, the Rustviz tool creates a graphical representation of it. Both single-threaded and multi-threaded Rust code can be utilized with the tool.

3.1 Installation and Environment Setup for RustViz

1. Install Rust and Cargo on your machine by running the command:
   curl https://sh.rustup.rs -sSf — sh
2. Clone the mdbook repository from GitHub onto your machine using the command git clone https://github.com/rust-lang/mdBook.
3. Navigate to the rustviz directory using the command cd rustviz.
4. Navigate to the mdbook directory using the command cd rustviz_mdbook
5. Run the command ./view_examples.sh to execute the script and view the examples.

   After doing above steps below type of output will display on terminal , Now open browser and navigate to http://localhost:8000/. We should be able to view the examples individually by selecting them from the left side bar. To view the visualization, click the toggle button on the top right corner of the code block.



```
(base) kavita@Kavitas-Air rustviz % cd mdbook
cd: no such file or directory: mdbook
(base) kavita@Kavitas-Air rustviz % cd rustviz_mdbook$
cd: no such file or directory: rustviz_mdbook$
(base) kavita@Kavitas-Air rustviz % cd rustviz_mdbook
(base) kavita@Kavitas-Air rustviz_mdbook % /view_examples.sh
zsh: no such file or directory: /view_examples.sh
(base) kavita@Kavitas-Air rustviz_mdbook % ./view_examples.sh
Generating visualizations for the following examples:
building copy...done
building func_take_ownership...done
building func_take_return_ownership...done
building function...done
building hatra1...done
building hatra1_test...done
building hatra2...done
building immutable_borrow...done
building immutable_borrow_lifetime...done
building immutable_borrow_method_call...done
building immutable_variable...done
building move_assignment...done
building move_different_scope...done
building move_func_return...done
building multiple_immutable_borrow...done
building mutable_borrow...done
building mutable_borrow_method_call...done
building mutable_variables...done
building nll_lexical_scope_different...done
building printing...done
building string_from_move_print...done
building string_from_print...done
building struct_lifetime...done
building struct_rect...done
building struct_rect2...done
building struct_string...done
building extra_credit...done
2023-05-11 11:03:55 [INFO] (mdbook::book): Book building has started
2023-05-11 11:03:55 [INFO] (mdbook::book): Running the html backend
2023-05-11 11:03:55 [INFO] (mdbook::book): Book building has started
2023-05-11 11:03:55 [INFO] (mdbook::book): Running the html backend
2023-05-11 11:03:55 [INFO] (mdbook::cmd::serve): Serving on: http://localhost:8000
2023-05-11 11:03:55 [INFO] (warp::server): Server::run; addr=[::1]:8000
2023-05-11 11:03:55 [INFO] (warp::server): listening on http://[::1]:8000
2023-05-11 11:03:55 [INFO] (mdbook::cmd::watch): Listening for changes...
2023-05-11 11:03:56 [INFO] (mdbook::cmd::serve): Files changed: ["/Users/kavita/rustviz/rustviz_mdbook/src/assets/func_take_ownership/vis_code.svg", "/Users/kavita/rustviz/rustviz_mdbook/src/assets/func_take_
ownership/source.rs", "/Users/kavita/rustviz/rustviz_mdbook/src/assets/immutable_borrow_lifetime/source.rs", "/Users/kavita/rustviz/rustviz_mdbook/src/assets/func_take_ownership/vis_timeline.svg"]
2023-05-11 11:03:56 [INFO] (mdbook::cmd::serve): Building book...
2023-05-11 11:03:56 [INFO] (mdbook::book): Book building has started
2023-05-11 11:03:56 [INFO] (mdbook::book): Running the html backend
```

3.2 Feature of Rustviz

Rustviz provides the following features:

1. **Graphical representation:** Rustviz provides a graphic representation of the code that enables us to view how to visualize the control flow of our code.
2. **Step-by-step execution:** The tool allows us to run the code step-by-step and examine how data is exchanged between variables and functions.
3. **Function tracing:** Rustviz enables us to track the execution of particular functions and see their calling and execution patterns.
4. **Multi-threaded support:** The tool can also be used to visualize and observe the interactions between threads in multi-threaded Rust programming.

3.3 How does Rustviz work :

RustViz demonstrates ownership and borrowing in Rust with a timeline of memory events for each variable in the source code that the teacher chooses to visualize. Rustviz works by parsing the Rust code and generating an intermediate representation. It then uses this intermediate representation to generate a graph that represents the control flow of the code.
Rustviz enables us to step-by-step run the code after the graph is produced. Rustviz highlights the current line of code as we go through the program and displays the variables' values and active function calls. This enables us to see how information is transferred between variables and functions.

3.4 Limitation of Rustviz

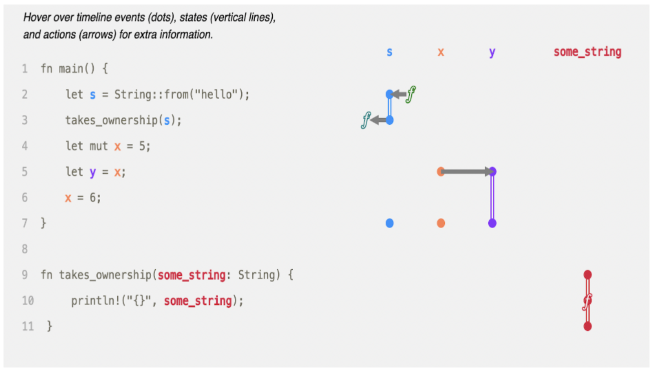
1. **No Verification:** The Rustviz tool only gives the visualization of the rust code. It doesn't verify any properties.

2. **Lack of Interactivity:** Rustviz provides a static visualization of code execution but lacks interactivity. It does not support stepping through the code or pausing at specific points to inspect variables.

3. **Limited Scope of Visualization:** Rustviz focuses on visualizing the control flow and state changes within a single function. It does not provide a comprehensive view of an entire program or support visualization across multiple functions or modules.

3.5 Rustviz by Example

1. Example 1

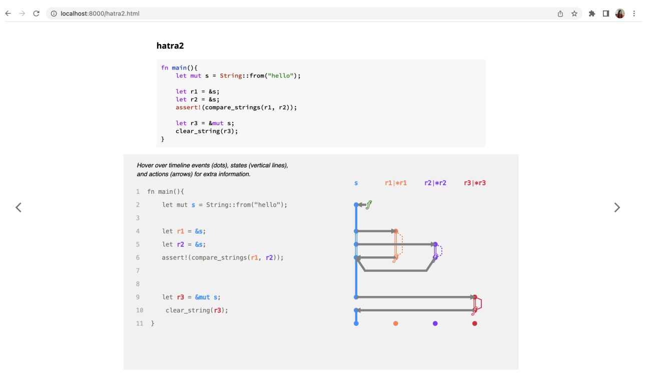


Hover over timeline events (dots), states (vertical lines), and actions (arrows) for extra information.

```
1  fn main() {
2      let s = String::from("hello");
3      takes_ownership(s);
4      let mut x = 5;
5      let y = x;
6      x = 6;
7  }
8
9  fn takes_ownership(some_string: String) {
10      println!("{}", some_string);
11  }
```

The above example shows a simple demonstration of moves, copies, and drops. Each resource has a unique owner in Rust, and when ownership changes, it is termed a move.
On line 2, String::from heap-allocates and returns a String, which moves ownership of that resource to s. 's' acquires ownership of a resource. The hollow blue Line Segment indicates that s cannot be reassigned or used to mutate the resource. After this move, s is no longer valid for use.
On Line 4, the x is assigned with the immutable integer value 5. We use let mut rather than let so that x can be reassigned. x acquires ownership of a resource.
On-line 5, initializing y with x. In Rust, classes like integers with stack-only data typically have an annotation called the Copy trait. These resources are copied rather than moved. x's resource is copied to y. 'x' is the resource's owner. The binding can be reassigned. 'y' is the resource owner. The binding cannot be reassigned.
The variables s, x, and y go out of scope at the end of the function on Line 7. Since x and y were owners, their resources were dropped. However, since s's resource was moved earlier, no drop occurred.

2. Example 2



localhost:8000/hatra2.html

hatra2

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

    let r1 = &s;
    let r2 = &s;
    assert!(compare_strings(r1, r2));

    let r3 = &mut s;
    clear_string(r3);
}
```

Hover over timeline events (dots), states (vertical lines), and actions (arrows) for extra information.

```
1  fn main(){
2      let mut s = String::from("hello");
3
4      let r1 = &s;
5      let r2 = &s;
6      assert!(compare_strings(r1, r2));
7
8
9      let r3 = &mut s;
10      clear_string(r3);
11  }
```

The above example demonstrates borrowing, i.e., working with references to resources. To construct an immutable reference in Rust, use the operator rather than the mut operator. This process is known as an immutable borrow.
On lines 4 and 5, s's resource is immutably borrowed. r1 and r2 immutably borrow a resource from s, and also cannot mutate *r1 and *r2.
On-line 6, r1 and r2 are passed to compare strings, which is shown in RustViz by the f symbol. Compare strings reads from r1 and r2. Since r1 and r2's borrows are no longer used after the function returns, the borrowed resource is returned to s.
On line 8, When creating a mutable reference to a resource in Rust, the mut operator is used rather than the operator. This process is known as mutable borrowing. s's resource is mutably borrowed. Showed r3's borrow is mutable in RustViz with a Solid Red Line and Can mutate the resource *r3 On Line 9, r3 is passed to clear string, and clear string is able to mutate the String through that reference. Return mutably borrowed resources from r3 to s. s's resource is no longer mutably borrowed.

4 Rust Verification by Using Kani Tool

Kani is an open-source verification tool designed to help developers prove properties about their Rust code. Kani uses model checking to check Rust programs and is particularly useful for verifying unsafe code blocks in Rust, where the unsafe code is unchecked by the Rust compiler. Developers can use Kani to prove various properties, including memory safety properties (e.g., null pointer dereferences, use-after-free), the absence of certain runtime errors (i.e., index out of bounds, panics), and the absence of certain types of unexpected behavior (e.g., arithmetic overflows). Kani can also verify custom properties provided in the form of user-specified assertions.

Kani's primary advantage is that it can detect bugs, corner cases, and unexpected behavior in Rust code, which can be missed by the Rust compiler. According to the Kani Rust Verifier blog, Kani is an open-source automated reasoning tool that proves the properties of Rust code. Kani has been used to check Rust code and has been shown to be effective in detecting bugs and other issues.

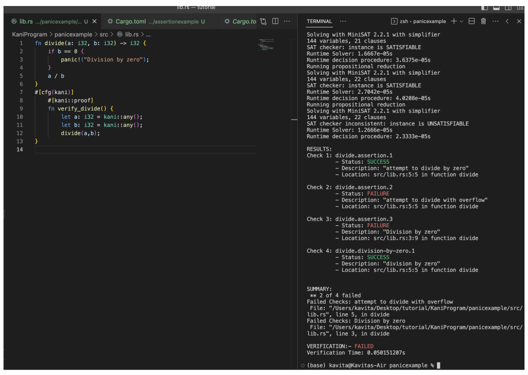
4.1 Installation and Environment Setup for Kani Tool

1. Install Rust and Cargo on your machine
2. Open a terminal and run the command: cargo install −−locked kani-verifier to install the Kani tool.
3. Run the command cargo kani setup to set up Kani for your Rust project.
4. Create a new Rust project using Cargo.
5. Add the dependency proptest = "1.0.0" to your Cargo.toml file.
6. Write your Rust program with Kani verification.
7. Run the command cargo kani from the terminal to verify your Rust program with Kani.
8. Kani will provide a result with any errors or bugs found in your program.
9. To get a visualization, run the command cargo kani −−visualize −−enable−unstable.
10. An HTML report will be generated, which will provide detailed information on where your program failed.

4.2 Example of Kani

1. **Panic Example**

   In the given code, the function divide takes two integers as input parameters and returns the result of dividing the first integer by the second integer. If the second input parameter is zero, the function panics with an error message "Div by zero". When Kani runs the verification, it checks for various kinds of errors and assertions. In this case, two assertions have failed:

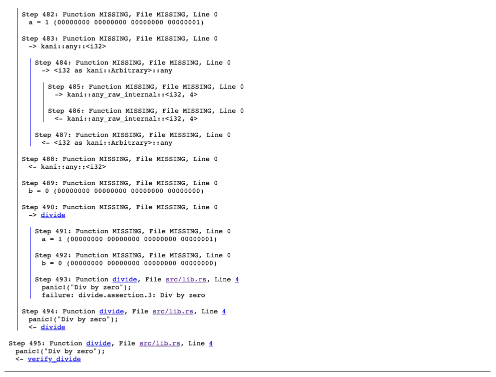


```
fn divide(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Division by zero");
    }
    a / b
}
#[cfg(kani)]
    #[kani::proof]
    fn verify_divide() {
        let a: i32 = kani::any();
        let b: i32 = kani::any();
        divide(a,b);
}
```

```
Solving with MiniSAT 2.2.1 with simplifier
144 variables, 21 clauses
SAT checker: instance is SATISFIABLE
Runtime Solver: 1.6667e-05s
Runtime decision procedure: 3.6375e-05s
Running propositional reduction
Solving with MiniSAT 2.2.1 with simplifier
144 variables, 22 clauses
SAT checker: instance is SATISFIABLE
Runtime Solver: 2.7042e-05s
Runtime decision procedure: 4.0208e-05s
Running propositional reduction
Solving with MiniSAT 2.2.1 with simplifier
144 variables, 22 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime Solver: 1.2666e-05s
Runtime decision procedure: 2.3333e-05s

RESULTS:
Check 1: divide.assertion.1
         - Status: SUCCESS
         - Description: "attempt to divide by zero"
         - Location: src/lib.rs:5:5 in function divide

Check 2: divide.assertion.2
         - Status: FAILURE
         - Description: "attempt to divide with overflow"
         - Location: src/lib.rs:5:5 in function divide

Check 3: divide.assertion.3
         - Status: FAILURE
         - Description: "Division by zero"
         - Location: src/lib.rs:3:9 in function divide

Check 4: divide.division-by-zero.1
         - Status: SUCCESS
         - Description: "division by zero"
         - Location: src/lib.rs:5:5 in function divide


SUMMARY:
 ** 2 of 4 failed
Failed Checks: attempt to divide with overflow
 File: "/Users/kavita/Desktop/tutorial/KaniProgram/panicexample/src/
lib.rs", line 5, in divide
Failed Checks: Division by zero
 File: "/Users/kavita/Desktop/tutorial/KaniProgram/panicexample/src/
lib.rs", line 3, in divide

VERIFICATION:- FAILED
Verification Time: 0.050151207s

(base) kavita@Kavitas-Air panicexample % 
```

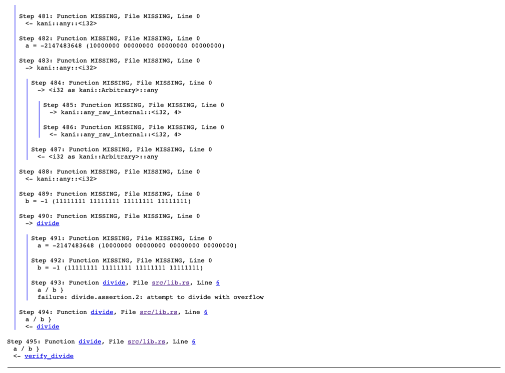
- **Attempt to divide with overflow**
  This is because the divide function does not check for integer overflow when performing the division operation. So, when a is a very large number and b is a very small number close to zero, an integer overflow can occur during the division operation, leading to this assertion's failure.

- **Div by zero**
  This is because the divide function can still panic with a "Div by zero" error message, even though we use panic to handle this case. Kani considers a panic to be a failure in the verification process, so this assertion fails.
  Below is the html report, where run trace can be found.
  Div by zero error report:



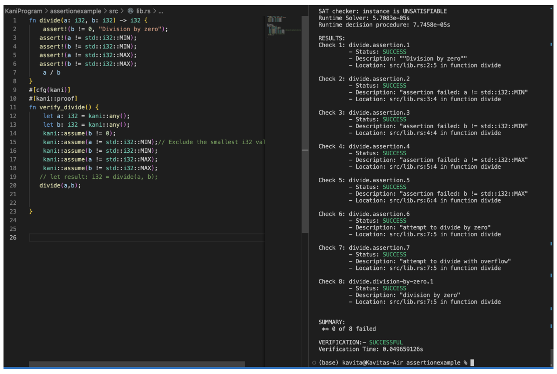
```
Step 482: Function MISSING, File MISSING, Line 0
  a = 1 (00000000 00000000 00000000 00000001)

Step 483: Function MISSING, File MISSING, Line 0
  -> kani::any::<i32>

  Step 484: Function MISSING, File MISSING, Line 0
    -> <i32 as kani::Arbitrary>::any

    Step 485: Function MISSING, File MISSING, Line 0
      -> kani::any_raw_internal::<i32, 4>

    Step 486: Function MISSING, File MISSING, Line 0
      <- kani::any_raw_internal::<i32, 4>

  Step 487: Function MISSING, File MISSING, Line 0
    <- <i32 as kani::Arbitrary>::any

Step 488: Function MISSING, File MISSING, Line 0
  <- kani::any::<i32>

Step 489: Function MISSING, File MISSING, Line 0
  b = 0 (00000000 00000000 00000000 00000000)

Step 490: Function MISSING, File MISSING, Line 0
  -> divide

  Step 491: Function MISSING, File MISSING, Line 0
    a = 1 (00000000 00000000 00000000 00000001)

  Step 492: Function MISSING, File MISSING, Line 0
    b = 0 (00000000 00000000 00000000 00000000)

  Step 493: Function divide, File src/lib.rs, Line 4
    panic!("Div by zero");
    failure: divide.assertion.3: Div by zero

Step 494: Function divide, File src/lib.rs, Line 4
  panic!("Div by zero");
  <- divide

Step 495: Function divide, File src/lib.rs, Line 4
  panic!("Div by zero");
  <- verify_divide
```

- **Attempt to divide with overflow**



```
Step 481: Function MISSING, File MISSING, Line 0
  <- kani::any::<i32>

Step 482: Function MISSING, File MISSING, Line 0
  a = -2147483648 (10000000 00000000 00000000 00000000)

Step 483: Function MISSING, File MISSING, Line 0
  -> kani::any::<i32>

  Step 484: Function MISSING, File MISSING, Line 0
    -> <i32 as kani::Arbitrary>::any

    Step 485: Function MISSING, File MISSING, Line 0
      -> kani::any_raw_internal::<i32, 4>

    Step 486: Function MISSING, File MISSING, Line 0
      <- kani::any_raw_internal::<i32, 4>

  Step 487: Function MISSING, File MISSING, Line 0
    <- <i32 as kani::Arbitrary>::any

Step 488: Function MISSING, File MISSING, Line 0
  <- kani::any::<i32>

Step 489: Function MISSING, File MISSING, Line 0
  b = -1 (11111111 11111111 11111111 11111111)

Step 490: Function MISSING, File MISSING, Line 0
  -> divide

  Step 491: Function MISSING, File MISSING, Line 0
    a = -2147483648 (10000000 00000000 00000000 00000000)

  Step 492: Function MISSING, File MISSING, Line 0
    b = -1 (11111111 11111111 11111111 11111111)

  Step 493: Function divide, File src/lib.rs, Line 6
    a / b }
    failure: divide.assertion.2: attempt to divide with overflow

Step 494: Function divide, File src/lib.rs, Line 6
  a / b }
  <- divide

Step 495: Function divide, File src/lib.rs, Line 6
  a / b }
  <- verify_divide
```

2. **Fixing the above error in Kani by using assume and assertion**
   The harness is written to ensure that the function can never panic or cause a division by zero or division with overflow errors. The function verify_divide uses Kani's kani::any() function to generate random



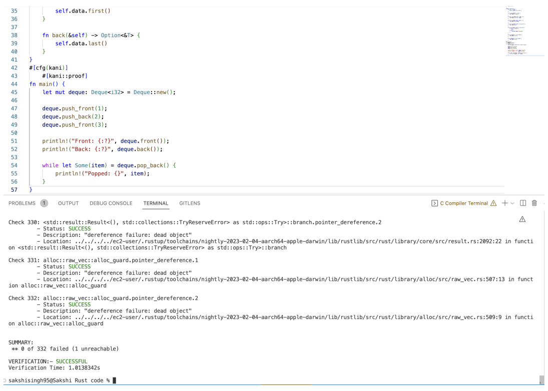
KaniProgram > assertionexample > src > lib.rs > ...

```rust
fn divide(a: i32, b: i32) -> i32 {
     assert!(b != 0, "Division by zero");
    assert!(a != std::i32::MIN);
    assert!(b != std::i32::MIN);
    assert!(a != std::i32::MAX);
    assert!(b != std::i32::MAX);
     a / b
}
#[cfg(kani)]
#[kani::proof]
fn verify_divide() {
     let a: i32 = kani::any();
     let b: i32 = kani::any();
     kani::assume(b != 0);
     kani::assume(a != std::i32::MIN);// Exclude the smallest i32 val
     kani::assume(b != std::i32::MIN);
     kani::assume(a != std::i32::MAX);
     kani::assume(b != std::i32::MAX);
    // let result: i32 = divide(a, b);
    divide(a,b);


}
```

```
SAT checker: instance is UNSATISFIABLE
Runtime Solver: 5.7083e-05s
Runtime decision procedure: 7.7458e-05s

RESULTS:
Check 1: divide.assertion.1
         - Status: SUCCESS
         - Description: ""Division by zero""
         - Location: src/lib.rs:2:5 in function divide

Check 2: divide.assertion.2
         - Status: SUCCESS
         - Description: "assertion failed: a != std::i32::MIN"
         - Location: src/lib.rs:3:4 in function divide

Check 3: divide.assertion.3
         - Status: SUCCESS
         - Description: "assertion failed: b != std::i32::MIN"
         - Location: src/lib.rs:4:4 in function divide

Check 4: divide.assertion.4
         - Status: SUCCESS
         - Description: "assertion failed: a != std::i32::MAX"
         - Location: src/lib.rs:5:4 in function divide

Check 5: divide.assertion.5
         - Status: SUCCESS
         - Description: "assertion failed: b != std::i32::MAX"
         - Location: src/lib.rs:6:4 in function divide

Check 6: divide.assertion.6
         - Status: SUCCESS
         - Description: "attempt to divide by zero"
         - Location: src/lib.rs:7:5 in function divide

Check 7: divide.assertion.7
         - Status: SUCCESS
         - Description: "attempt to divide with overflow"
         - Location: src/lib.rs:7:5 in function divide

Check 8: divide.division-by-zero.1
         - Status: SUCCESS
         - Description: "division by zero"
         - Location: src/lib.rs:7:5 in function divide


SUMMARY:
 ** 0 of 8 failed

VERIFICATION:- SUCCESSFUL
Verification Time: 0.049659126s

(base) kavita@Kavitas-Air assertionexample %
```

inputs a and b. Then the kani::assume() function adds preconditions to the input a and b that exclude the smallest and largest values of i32 and also exclude 0 for b. These preconditions are designed to avoid the panics or division by zero or overflow that can happen when these specific inputs are used. Finally, the divide function is called with the generated inputs a and b.
Assert!: This macro is used within the divide function to check if certain conditions are met before performing the division operation. If the condition specified

3. **Double-ended Queue**

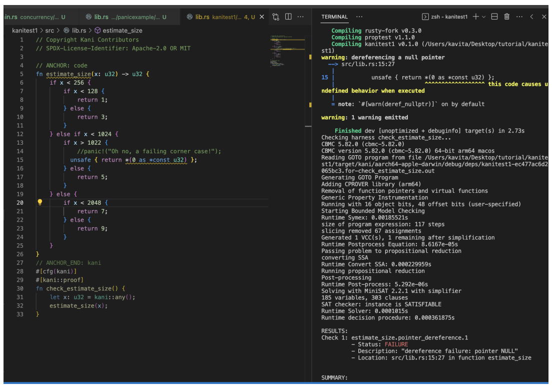


```rust
        self.data.first()
    }

    fn back(&self) -> Option<&T> {
        self.data.last()
    }
}
#[cfg(kani)]
    #[kani::proof]
fn main() {
    let mut deque: Deque<i32> = Deque::new();

    deque.push_front(1);
    deque.push_back(2);
    deque.push_front(3);

    println!("Front: {:?}", deque.front());
    println!("Back: {:?}", deque.back());

    while let Some(item) = deque.pop_back() {
        println!("Popped: {}", item);
    }
}
```

```
Check 330: <std::result::Result<(), std::collections::TryReserveError> as std::ops::Try>::branch.pointer_dereference.2
         - Status: SUCCESS
         - Description: "dereference failure: dead object"
         - Location: ../../../../ec2-user/.rustup/toolchains/nightly-2023-02-04-aarch64-apple-darwin/lib/rustlib/src/rust/library/core/src/result.rs:2092:22 in functi
on <std::result::Result<(), std::collections::TryReserveError> as std::ops::Try>::branch

Check 331: alloc::raw_vec::alloc_guard.pointer_dereference.1
         - Status: SUCCESS
         - Description: "dereference failure: dead object"
         - Location: ../../../../ec2-user/.rustup/toolchains/nightly-2023-02-04-aarch64-apple-darwin/lib/rustlib/src/rust/library/alloc/src/raw_vec.rs:507:13 in funct
ion alloc::raw_vec::alloc_guard

Check 332: alloc::raw_vec::alloc_guard.pointer_dereference.2
         - Status: SUCCESS
         - Description: "dereference failure: dead object"
         - Location: ../../../../ec2-user/.rustup/toolchains/nightly-2023-02-04-aarch64-apple-darwin/lib/rustlib/src/rust/library/alloc/src/raw_vec.rs:509:9 in functi
on alloc::raw_vec::alloc_guard


SUMMARY:
 ** 0 of 332 failed (1 unreachable)

VERIFICATION:- SUCCESSFUL
Verification Time: 1.0138342s

sakshisingh95@Sakshi Rust code %
```

The code defines a Deque struct that holds a vector (data) of elements of type T. This represents a double-ended queue. In the main() function, a deque of type Deque(i32) is created, and various operations are performed on it. Three elements are inserted into the deque using push_front() and push_back(). The pop_back() method is called in a loop until the deque is empty, and each popped item is printed. The code passed the verification process with only internal standard library checks resulting in failures. This suggests that the code itself is correct and satisfies the properties being checked by the Kani tool. There is one property that is unreachable because the code is no longer called.

4. **Dereference failure**



kanitest1 > src > lib.rs > estimate_size

```rust
// Copyright Kani Contributors
// SPDX-License-Identifier: Apache-2.0 OR MIT

// ANCHOR: code
fn estimate_size(x: u32) -> u32 {
    if x < 256 {
        if x < 128 {
            return 1;
        } else {
            return 3;
        }
    } else if x < 1024 {
        if x > 1022 {
            //panic!("Oh no, a failing corner case!");
          unsafe { return *(0 as *const u32) };
        } else {
            return 5;
        }
    } else {
        if x < 2048 {
            return 7;
        } else {
            return 9;
        }
    }
}
// ANCHOR_END: kani
#[cfg(kani)]
#[kani::proof]
fn check_estimate_size() {
    let x: u32 = kani::any();
    estimate_size(x);
}
```

```
   Compiling rusty-fork v0.3.0
   Compiling proptest v1.1.0
   Compiling kanitest1 v0.1.0 (/Users/kavita/Desktop/tutorial/kanite
st1)
warning: dereferencing a null pointer
  --> src/lib.rs:15:27
   |
15 |             unsafe { return *(0 as *const u32) };
   |                             ^^^^^^^^^^^^^^^^^^ this code causes u
ndefined behavior when executed
   |
   = note: `#[warn(deref_nullptr)]` on by default

warning: 1 warning emitted

    Finished dev [unoptimized + debuginfo] target(s) in 2.73s
Checking harness check_estimate_size...
CBMC 5.82.0 (cbmc-5.82.0)
CBMC version 5.82.0 (cbmc-5.82.0) 64-bit arm64 macos
Reading GOTO program from file /Users/kavita/Desktop/tutorial/kanite
st1/target/kani/aarch64-apple-darwin/debug/deps/kanitest1-ec477ac6d2
065bc3.for-check_estimate_size.out
Generating GOTO Program
Adding CPROVER library (arm64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 16 object bits, 48 offset bits (user-specified)
Starting Bounded Model Checking
Runtime Symex: 0.00185521s
size of program expression: 117 steps
slicing removed 67 assignments
Generated 1 VCC(s), 1 remaining after simplification
Runtime Postprocess Equation: 8.6167e-05s
Passing problem to propositional reduction
converting SSA
Runtime Convert SSA: 0.000229959s
Running propositional reduction
Post-processing
Runtime Post-process: 5.292e-06s
Solving with MiniSAT 2.2.1 with simplifier
185 variables, 303 clauses
SAT checker: instance is SATISFIABLE
Runtime Solver: 0.0001015s
Runtime decision procedure: 0.000361875s

RESULTS:
Check 1: estimate_size.pointer_dereference.1
         - Status: FAILURE
         - Description: "dereference failure: pointer NULL"
         - Location: src/lib.rs:15:27 in function estimate_size


SUMMARY:
```

In the provided code, the estimate_size function is used to estimate the size of a given value x based on certain conditions. The function uses conditional statements to determine the size, and in one of the cases, it uses an unsafe block to return an arbitrary value. The pointer dereference error occurs when x is greater than 1022. In that case, the code attempts to dereference a null pointer by using the unsafe block. This is a serious issue that can cause undefined behavior, including crashes or security vulnerabilities.

The Kani tool can detect this issue because it analyzes the code and generates input values that exercise all the possible code paths. In this case, Kani can generate an input value for x that is greater than 1022, causing the code to execute the unsafe block and result in a pointer dereference error. Kani then reports this error as a failing assertion in the proof harness.

5 Conclusion

In conclusion, our project aimed to gain familiarity with the Rust programming language, specifically focusing on its unique features such as ownership and borrowing. We also explored two tools, RustViz and Kani, that are crucial in understanding and verifying Rust code. RustViz proved to be an invaluable resource in our journey, providing interactive visualizations that helped us grasp the intricacies of ownership and borrowing in Rust. By generating visual representations from simple Rust programs, RustViz enhanced our understanding of how memory is managed and shared in Rust, ultimately improving our ability to write efficient code. However, when it came to verifying Rust code, we recognized the need for a more specialized tool. This is where Kani came into play as a bit-precise model. It helped us understand unsafe rust, and it also provided a detailed report, including visualization. Also, Kani exercises all possible code path and find out the bug which can be missed by the compiler. We learned examples like panic, unsafe, dereference failure, double-ended queue, assume and assertion etc.

6 Our Github Link to access Code

Github repository

7 Reference

1. Rust Book
2. Rust Github
3. Papers on Rust Language
4. Kani Github
5. Kani book
6. Rustviz Github
7. Rustviz Paper