Formal Verification for Rust Ported Codebases

Introduction

1. Problem Statement:

- **Original Statement:**
  - Porting code from C/C++ to Rust is a growing trend in systems programming due to how Rust offers Memory and Concurrency Safety. One example is the Rust For Linux project that adds Rust as a second programming language to C for writing kernel components.
  - We aim to employ formal verification methods, including model checking and symbolic execution, to ensure the correctness of memory safety properties and functional correctness of certain properties across the C/C++ and ported Rust code
- **POPL Angle:**
  - **Integration of the best aspects of the two programming languages**: The project aims at porting C codebases to Rust to achieve the memory safety guarantees provided by Rust, while making minimal changes to the legacy code.
  - **Exploring different program paths without concrete inputs**: This is a common challenge in software testing. Using formal verification techniques, we verified code with different programming language concepts like:
    - Memory Safety: Rust prevents memory-related errors like null pointer dereferences, and buffer overflows through its ownership system.
    - Ownership and Borrowing: Rust's ownership system ensures exclusive data ownership, preventing data races, while borrowing allows safe temporary access.
    - Zero-Cost Abstractions: Rust allows expressive high-level abstractions without sacrificing runtime performance.
    - Type System and Static Typing: Rust's strong type system catches errors at compile time, improving code reliability.
- **Previous Solutions:**
  - Symbolic execution and testing tools like KLEE are well-established in formal verification, but they have rarely been used to verify hybrid Rust and C/C++ codebases
  - Some papers which discuss similar problems can be found here and here
- **Differences in Your Solution:**

- The unique aspect here is the application of symbolic execution to hybrid Rust and C/C++ code snippets involving memory operations in the heap and in global uninitialized data space.

2. Software Architecture:

- **Architecture Overview:**
  - Kani Rust Verifier: simple linked list in Rust to check for null pointer dereferences
  - KLEE: custom_malloc code to check for memory_out_of_bounds errors. There is also a hybrid code that consists of the linked_list implemented in Rust and the append function implemented in C, which is called in the Rust code through FFI

3. POPL Aspects:

Note: We have given references along with an explanation to the specific code lines for the given aspects.

- **Memory Safety**: We try to take advantage of Rust's ownership, borrowing and lifetime system to ensure memory safety in the ported codebase, thereby mitigating common memory-related pitfalls like null pointer dereferencing and memory leaks, which are usually problems in a pure C codebase. Our examples use a linked list implemented in Rust to take advantage of its memory safety guarantees.

- **Symbolic Execution**: We use KLEE, an LLVM-based symbolic execution engine, to test the C code for memory out-of-bounds errors. We use the functions `klee_make_symbolic` and `klee_assume` to create symbolic data and to assume certain conditions on the symbolic data to guide the symbolic execution. We are also using klee_check_memory_access to check for memory out-of-bounds errors.

- **Model Checking**: We are using Kani, a model checker for Rust, to check for null pointer dereferences in the Rust code. Kani specializes in checking memory unsafe Rust code. This is important for us since we are trying to port C code to Rust, and we want to ensure that the Rust code, which includes `unsafe` blocks, is memory-safe.

- **Hybrid Linking**: We use FFI to link the Rust code with the C code. We use the `#[no_mangle]` attribute to ensure the compiler does not mangle the function name. We also use the `extern` keyword to indicate that the function is implemented outside of Rust.

- **Choice of Programming Paradigms**: Implementation of a certain functionality in Rust or C/C++ is decided based on the programming paradigm that is best suited for the task. Each programming language has pros and cons; in our use case, Rust offers memory safety. Still, many codebases have not been converted to Rust, which will take a lot of time and require a lot of testing to ensure that the functionality is preserved. On the other hand, C is widely used in many codebases and is a low-level language that is not memory-safe.

- **Data Structures**: In our examples, we are testing it on a linked list implementation, which we have written in Rust. In many codebases, data structures are crucial to the code and are often included in header files written in C, the parent language here. We should include the Rust-implemented linked list inside the C code to test whether we can use data structures implemented in Rust in C codebases.

- **Object-Oriented Programming Principles**: We have implemented a linked list according to the object-oriented programming principles in Rust. We have used structs to encapsulate the data and functions to operate on the data. We have also used the `impl` keyword to implement the functions.

4. Results and Testing:

- **Preliminary Tests Conducted:**
  - We used the test.rs file to check whether Kani detects a null pointer dereference in the code.
  - We used the hybrid folder in the code external directory to test the linking of the Rust code with the C code.

- **KLEE tests:**
  - We implemented a custom_malloc function in C to check whether KLEE detected memory out-of-bounds errors. We also aim to apply it in future work to combine Rust and C LLVM bitcode to verify hybrid code.

5. Potential for Future Work Given More Time:

- **Extended Features:**

1. **Enhanced Symbolic Execution:**
   - Investigate advanced features in symbolic execution engines like KLEE. We want to compile hybrid Rust code into LLVM bitcode while also linking all the necessary libraries not supported by LLVM Linker.

2. **Integration with Larger Codebases:**
   - Extend the symbolic testing approach to larger and more complex codebases to evaluate scalability and applicability and collaborate with the other group to test ported real-world code.

3. **Property-Based Testing Scope:**
   - Extend the project's scope to support more property-based testing, enabling the specification of properties to be checked during symbolic execution.

4. **Benchmarking and Evaluation:**
   - Establish a comprehensive benchmark suite and evaluate the performance of the symbolic execution engine on various types of programs and real-world use cases.

Model Checking in Rust

Overview

This Rust code provides a basic implementation of a singly linked list. The linked list consists of nodes, each containing an integer value (`i32`) and a pointer to the next node in the list. The implementation includes methods for creating a new node, initializing a linked list, appending nodes to the end of the list, and printing the list elements.

Code Structure

Node Struct

The `Node` struct represents a single node in the linked list. It has two fields:

- `data` : An integer value associated with the node.
- `next` : A mutable raw pointer (`*mut Node`) pointing to the next node in the list. We used mutable raw pointers so that we could introduce unsafe operations.

LinkedList Struct

The `LinkedList` struct represents the entire linked list and contains a mutable raw pointer to the head of the list. It includes methods for initializing an empty linked list (`new`), appending a node to the end of the list (`append`), and printing the elements of the list (`print`).

Functions

- `Node::new(data: i32) -> Self` : A constructor for creating a new node with the given integer value.
- `LinkedList::new() -> Self` : A constructor for creating a new, empty linked list.
- `LinkedList::append(&mut self, data: i32)` : Appends a new node with the specified data to the end of the linked list.
- `LinkedList::print(&self)` : Prints the elements of the linked list.

do_stuff Function

The `do_stuff` function demonstrates the usage of the linked list by creating an instance of `LinkedList` , appending three nodes with values 1, 2, and 3, and then printing the elements of the list. We are checking this function using Kani for null pointer dereferences. The code uses unsafe Rust features due to the manual management of raw pointers. In real code, care must be taken to ensure that operations involving raw pointers are safe and do not lead to memory safety issues.

Building and Running

To build and run the code, ensure a cargo environment is set up. Then, compile and execute the code using:

```
cargo run
```

Verification

The code includes a verification block using the Kani-proof system. The `verify_success` function calls the `do_stuff` function. Kani provides a simple proof of null pointer dereference.

Notes

- This implementation focuses on simplicity and may not handle all edge cases or optimizations.

Symbolic Execution Test in C

Overview

This C code snippet utilizes the KLEE symbolic execution engine to test a program that involves symbolic data. The program defines a structure `data` containing an offset and an array of integers. The goal is to perform symbolic execution on this program to explore bounds checking for an array in C using KLEE.

Code Structure

Data Structure

The `data` structure includes:

- `offset` : An unsigned integer indicating an offset.
- `array` : An array of unsigned integers with a maximum size of `MAX_SIZE` .

Memory

The code reserves a block of memory (`data_memory`) in uninitialized global data space to be used for symbolic execution.

Functions used

- `check(void* start, unsigned int size_bytes, const char* name)` : A helper function to create symbolic data within a specified range and test for memory out of bounds.
- `custom_malloc(int size)` : This function attempts to allocate at `size * block` location of data_memory, where block is of size 64 bytes. Successful execution returns a pointer to the starting address of the allocated memory.
- `int main()` : The `custom_malloc` function is called for a given `size` . On **success**, The program performs a conditional check based on the symbolic data. If the value at a specific offset in the `array` is equal to the ASCII value of 'a', it prints "true"; otherwise, it prints "false."

Assumptions

The code includes an assumption (`klee_assume`) on the `offset` field to limit its possible values. This is a common technique in symbolic execution to reduce the state space and make the analysis more manageable. Exploding state space can be a real problem with large codebases for symbolic execution.

[FAILED ATTEMPT] Custom Malloc For KLEE Verification(custom_malloc.c)

Why did this fail, and why are we documenting this

We attempted to make a `custom malloc` code for KLEE Verification. However, after completing the code and documenting it fully, we realised that the code was incorrect due to some starting assumptions we made. This can be fixed. However, we decided it was better to write a newer code from scratch to ensure we could complete it within the deadline. Below is the documentation we had written for the code.

Overview

- To enable verification with tools like KLEE, the code necessitates memory allocation on the stack rather than the heap. The conventional malloc function, which allocates memory on the heap and returns its pointer isn't compatible with this verification process.
- To address this, a custom malloc function has been written by us. This implementation allocates memory within a global buffer, aiming to replicate the memory allocation behaviour of the original malloc. The change is minimal for developers—simply substituting custom_malloc for malloc when requesting memory. The specifics of how and where memory is obtained remain opaque to the developer, akin to malloc. KLEE can successfully work on this global buffer.
- These adjustments are crucial to ensure compatibility with verification tools. Adhering to specific coding practices and patterns significantly enhances the effectiveness of such tools in verification,

contributing to the code's overall correctness and safety for deployment in production environments.

Code Logic Progression

This code snippet demonstrates a simplistic memory allocation scheme using a fixed-size buffer and manual bookkeeping of allocated regions. It also introduces symbolic execution to explore different execution paths typically used in testing and verification

1. Allocate memory from `globalBuffer`.
2. Assign symbolic values to allocated memory blocks.
3. Print these symbolic values.
4. Free allocated memory.

Code Structure

BUFFER_SIZE defines the size of the global buffer

globalBuffer is an array of characters acting as a global memory buffer with a fixed size

MemoryAllocation Struct Represents a memory allocation with two fields:

- `start address` : Pointer to the start of the allocated memory
- `size` : Size of the allocated memory block.

allocatedRegions Array

- An array of `MemoryAllocation` structures to store information about allocated memory regions.
- `numAllocatedRegions` : Keeps track of the number of allocations.

globalMalloc Function

- Takes a size argument and attempts to allocate memory from `globalBuffer`.
- Checks if there's enough space and assigns memory from `globalBuffer`.
- Stores information about the allocated memory in `allocated regions`.
- Notice how klee_make_symbolic has been used here to keep track of memory locations has been used here

globalFree Function

- Frees previously allocated memory by matching the pointer to an allocated region.
- Clears the corresponding entry in `allocated regions`.

Main Function

- Memory Allocation and Initialization:
  - Calls `globalMalloc` twice to allocate memory for two dynamic arrays (`dynamicArray1` and `dynamicArray2`)
  - Assigns symbolic values to these arrays using klee_make_symbolic. This is typically used in symbolic execution to explore different paths in the code without knowing exact concrete values.
- Printing Array Values:
  - prints the values of `dynamicArray1` and `dynamicArray2` .
- Freeing Allocated Memory:
  - Calls globalFree to free the memory allocated for `dynamicArray1` and `dynamicArray2`