How can we formally verify Rust for Linux?

Alastair Reid
@alastair_d_reid

Google Research

https://project-oak.github.io/rust-verification-tools/

The verification continuum

Google Research

https://project-oak.github.io/rust-verification-tools/

Agenda

- What code to verify?
- What properties to verify?
- How can we use tools we have today?
- What needs fixed before this is viable?

Google Research

https://project-oak.github.io/rust-verification-tools/

Spoiler: It is not usable yet

- Tool problems
- I didn't find bugs
- I didn't verify anything
- (I am now working on a new, unrelated project)

Google Research

https://project-oak.github.io/rust-verification-tools/

Detailed blog post https://project-oak.github.io/rust-verification-tools/

Using KLEE on Rust-for-Linux (part 3)

Aug 24, 2021

…Linux, we are going to write a simple
…re device driver written in Rust and
…EE. It's worth repeating that the goal of
…or to find bugs in it. Instead, my goal is
…that verification yourself. You might
…st by creating better mocks and test
…ifferent answers to the questions in the
…KLEE that I described in the second
…in this branch of my fork of Rust for

…harness like this

```
…ev;
…re>, FileState>(registration)?;
```

Using KLEE on Rust-for-Linux (part 2)

Aug 23, 2021

…ded model checkers with Linux
…s series on using KLEE on the
…This second part, digs deeply
…ed tools like KLEE. (Warning: it
…ted in.) The final part will show
…described in this post are in this

…LLVM bitcode files.

…y the Rust for Linux

…de.

…we will just focus on building
…E to symbolically execute the

…a few of them are a bit slow.
…ds into your shell in the

Using KLEE on Rust-for-Linux (part 1)

Aug 22, 2021

The Rust for Linux project is working on adding support for the Rust language to the Linux kernel with the hope that using Rust will make new code more safe, easier to refactor and review, and easier to write. (See the RFC for more detail about goals and for the varied responses of the Linux Kernel community.)

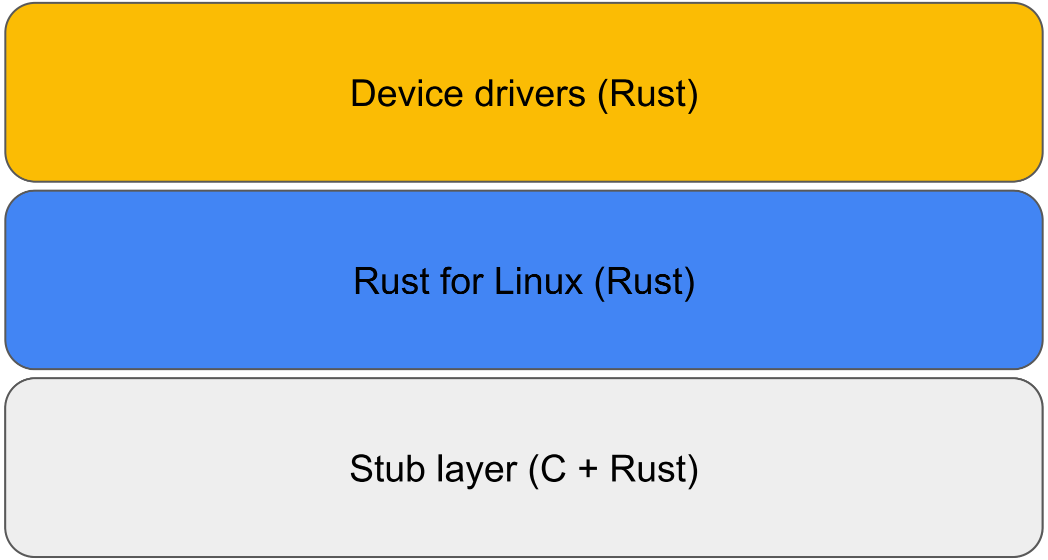
Back in April, I took a look at whether we could use our Rust verification tools on the Rust for Linux repo to provide further safety. Most of our work is based on the KLEE symbolic execution tool and I was able to get that to work. For *reasons*, I did not get to explore this very deeply after that but I thought it would be useful to describe what I was able to do and some of the questions raised by the work as a guide to how you might tackle the problem in the future.

I have split this blog into three parts because it was getting quite long. In this part, I'll start by looking at some key questions around what properties and code we want to check. The second part, will dive deeply into how to build Rust-for-Linux in a way that you can use KLEE on it. (Many people will want to skip this part.) And the final part, will return to the questions by creating test harnesses and stubs that could be used to check the Rust-for-Linux code for bugs.

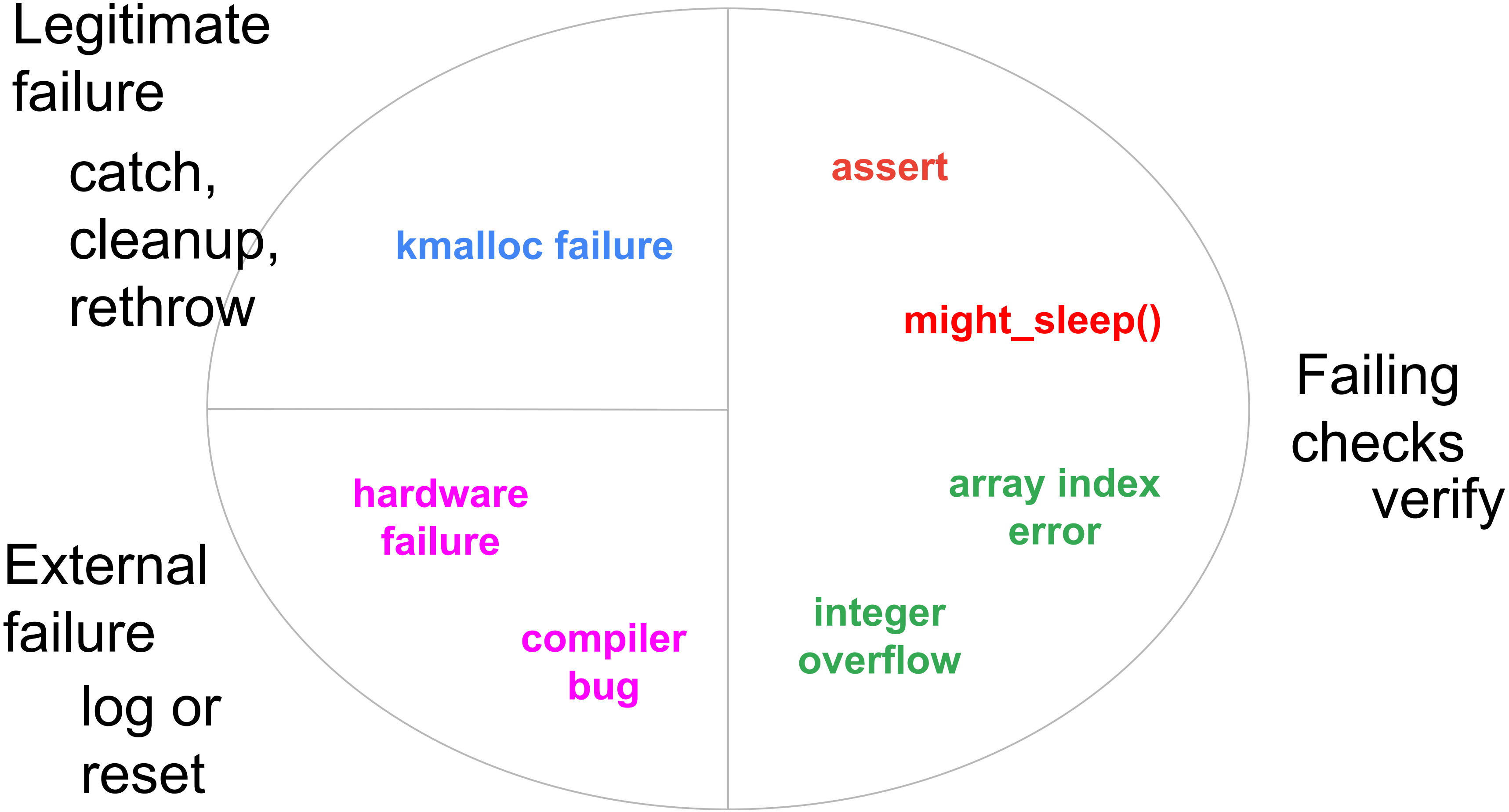
As with the previous post on using KLEE with CoreUtils, my goal in this post is to help others to use tools from the formal verification community to check code like this rather than to do that checking myself. In particular, I will not find any bugs, I will not attempt to provide evidence that this is worth doing and I will not create a verification system that is ready to integrate into any project. These (and other limitations listed at the end of the last post) all need to be fixed before I would recommend that you try to use these tools as part of your regular workflow. But, I hope that this series will give you an

Google Research

What code to verify?

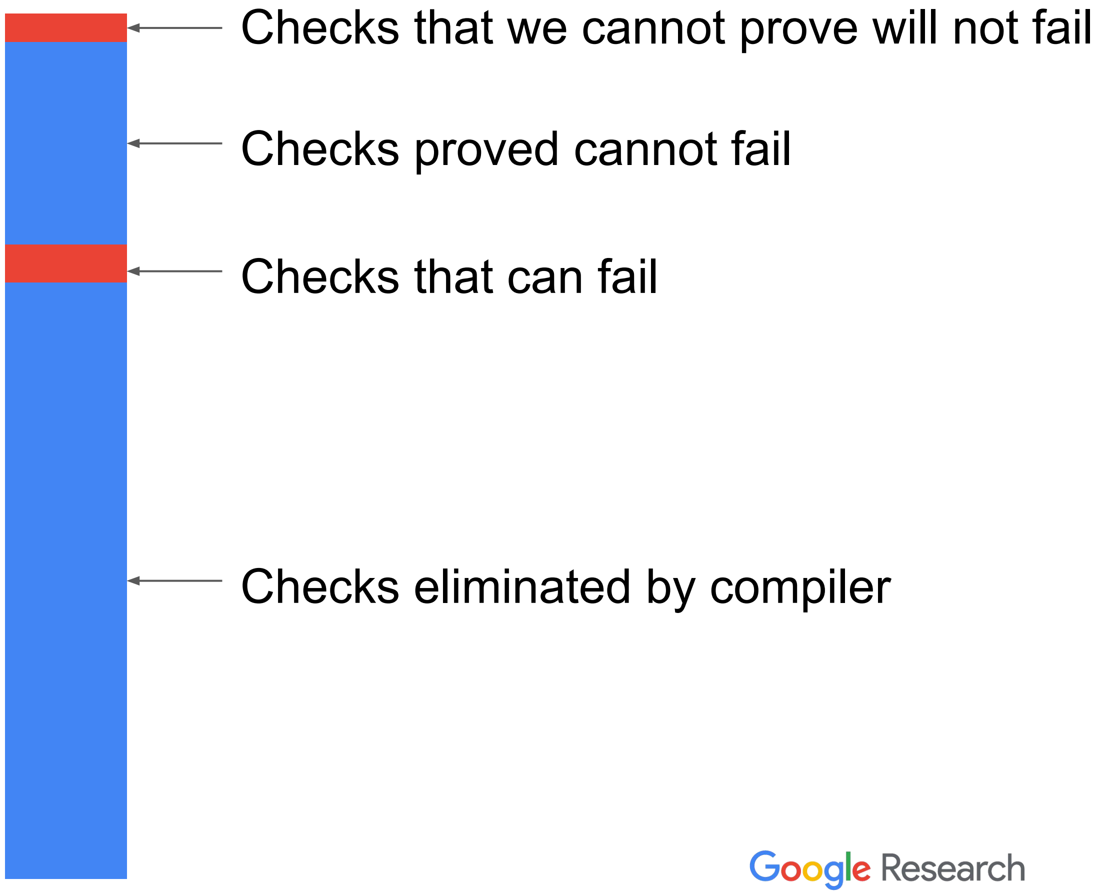


https://project-oak.github.io/rust-verification-tools/


Legitimate failure
catch, cleanup, rethrow
kmalloc failure
assert
might_sleep()
Failing checks
verify
array index error
integer overflow
hardware failure
compiler bug
External failure
log or reset

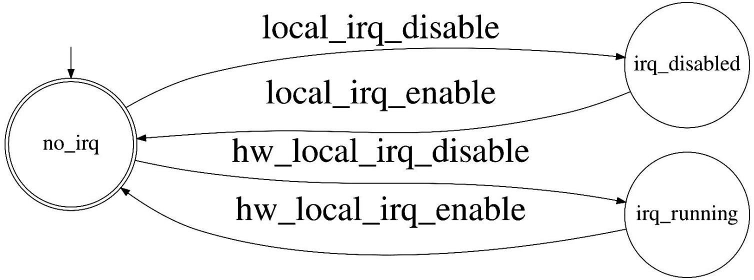
#1

Programmer & Compiler inserted checks



Google Research

#2: State machines



- Many state machines in OS
  - Kernel, modules, devices, objects, etc.
- Check that state m/c changes are allowed
- See [Formal verification made easy and fast]() (LPC 2019)

Google Research

Diagram: Daniel Bristot de Oliviera et al, “Untangling the Intricacies of Thread Synchronization in the PREEMPT_RT Linux Kernel”, ISORC 2019.

#3: System invariants

- Fast systems code has many invariants
- Executable invariants
  - assertions on function entry/exit

https://project-oak.github.io/rust-verification-tools/

Google Research

#4: Functional correctness

1. Write a formal specification of your code
2. Verify code against the specification
3. Update specification as code changes

Google Research

https://project-oak.github.io/rust-verification-tools/

Writing verification harnesses (parameterized tests)

1. Write a test of your code using fixed values
2. Replace fixed values with parameters
   a. Random values → fuzzing
   b. Symbolic values → formal verification
3. Profit: one test that can be used in two ways

Google Research

https://project-oak.github.io/rust-verification-tools/

A concrete test

```
#[test]
fn test_fileops2() -> Result<()> {
    let registration = &RustSemaphore::init()?._dev;
    let file_state = *mk_file_state::<Arc<Semaphore>, FileState>(registration)?;
    let file = File::make_fake_file();

    test_write(&file_state, &file, 42);
    test_read(&file_state, &file, 6);
    Ok(())
}
```

Google Research

https://project-oak.github.io/rust-verification-tools/using-propverify/

PropTest

What I want to write…

```
proptest! {
    #[test]
    fn test_fileops2(wlen in 0..=1000usize, rlen in 0..=1000usize) -> Result<()> {
        let registration = &RustSemaphore::init()?._dev;
        let file_state = *mk_file_state::<Arc<Semaphore>, FileState>(registration)?;
        let file = File::make_fake_file();

        test_write(&file_state, &file, wlen);
        test_read(&file_state, &file, rlen);
        Ok(())
    }
}
```

https://project-oak.github.io/rust-verification-tools/using-propverify/

Google Research

PropTest and PropVerify

What I want to write…

```
proptest! {
    #[test]
    fn test_fileops2(wlen in 0..=1000usize, rlen in 0..=1000usize) -> Result<()> {
        let registration = &RustSemaphore::init()?._dev;
        let file_state = *mk_file_state::<Arc<Semaphore>, FileState>(registration)?;
        let file = File::make_fake_file();

        test_write(&file_state, &file, wlen);
        test_read(&file_state, &file, rlen);
        Ok(())
    }
}
```

https://project-oak.github.io/rust-verification-tools/using-propverify/

Google Research

What works today

```
#[no_mangle]
pub fn test_fileops2() -> Result<()> {
    let wlen = AbstractValue::abstract_value();
    let rlen = AbstractValue::abstract_value();

    let registration = &RustSemaphore::init()?._dev;
    let file_state = *mk_file_state::<Arc<Semaphore>, FileState>(registration)?;
    let file = File::make_fake_file();

    test_write(&file_state, &file, wlen);
    test_read(&file_state, &file, rlen);
    Ok(())
}
```

https://project-oak.github.io/rust-verification-tools/2021/08/24/rust-on-linux-3.html

How to run a verification tool

1. Write a parameterized test
2. Write stub functions for C code called from R4L
3. Generate LLVM bitcode: WLLVM, --emit=llvm-bc
4. Link bitcode files
5. Run verification tool (KLEE)

Google Research

https://project-oak.github.io/rust-verification-tools/

Tool issues today (many of these are changing)

1. Cargo integration → couldn't use PropVerify
2. KLEE only for now → finding bugs, not proving
3. LLVM11 vs LLVM12
4. No concurrency support

Google Research

https://project-oak.github.io/rust-verification-tools/

Summary

- What code to verify
- What properties to verify

  Compiler inserted checks, state machines, system invariants, …
- Parameterized tests
  - Verification continuum (PropVerify and PropTest)
- Tool issues – changing fast

Google Research

https://project-oak.github.io/rust-verification-tools/

Thank You

Alastair Reid

@alastair_d_reid

Google Research