Using KLEE with large Rust programs

Alastair Reid and Shaked Flur
{adreid, sflur}@google.com
@alastair_d_reid, @FlurShaked

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

Google Research

KLEE workshop, 10–11 June 2021

Agenda

01 Why large Rust programs?

02 High/low-level APIs for KLEE

03 The challenge: so many features

04 Experience

05 Conclusions

Google Research

Why large Rust programs?

Fearless security: memory & thread safety
Significant momentum & support

Rust programs use lots of libraries
Many verification approaches being explored

→ MIR → LLVM → x86

Google Research

Oxidizing the KLEE API: low level

KLEE’s C API	Idiomatic Rust API
`klee_make_symbolic`	`trait T::abstract_value`
`klee_assume`	`fn verifier::assume`
`klee_abort`	`fn verifier::abort`
`klee_silent_exit`	`fn verifier::reject`
`klee_get_value_`*`ty`*	`trait T::get_concrete_value`
`klee_is_symbolic`	`trait T::is_symbolic`
`klee_{open,close}_merge`	`macro verifier::coherent!`

Google Research

To fuzz or to verify – Why not both?

Goal: common API for fuzzing and for DSE, BMC, …

- Based on Rust property-based testing library proptest
- (Similar to Python's Hypothesis library)

"`proptest!{...}`" expands to suitable test harness

EDSL for creating structured symbolic values

- "`0..1000`"
- "`arbitrary_ascii(10)`"
- "`[0..1000; 3]`", "`(0..1000, arbitrary_ascii(10))`"

Fuzzing / DSE test harness

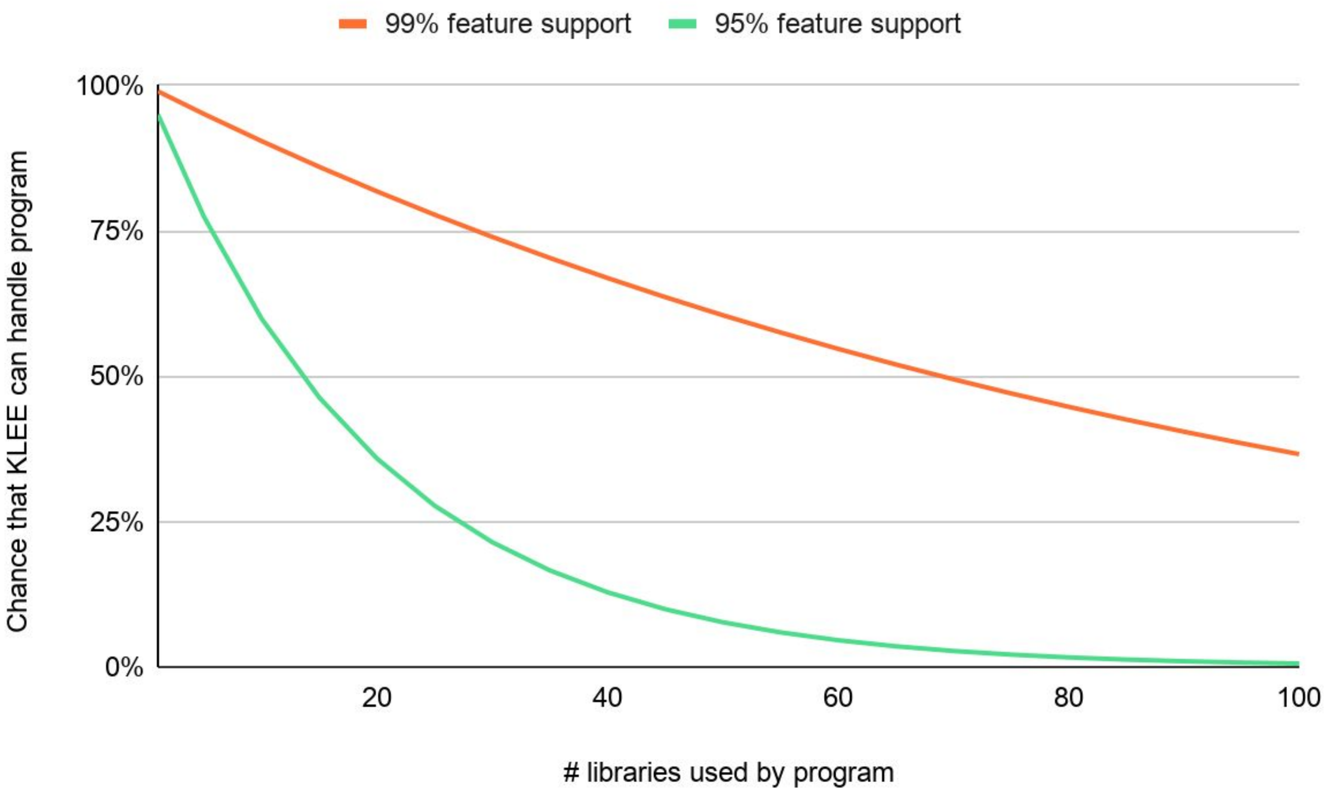
Symbolic value EDSL

DSE test harness

```
proptest! {
    #[test]
    fn multiply(a in 1..=1000u32, b in 1..=1000u32) {
        let r = a*b;
        assert!(1 <= r && r <= 1000000);
    }
}
```

(Overly simple example)

The challenge: so many features



Google Research

The challenge: so many features

Language features: tuples, closures, traits, ...

Compiler features: fshl/fshr, LLVM-11

Runtime features: concurrency

C-Rust interoperation: FFI-calls, linking, ...

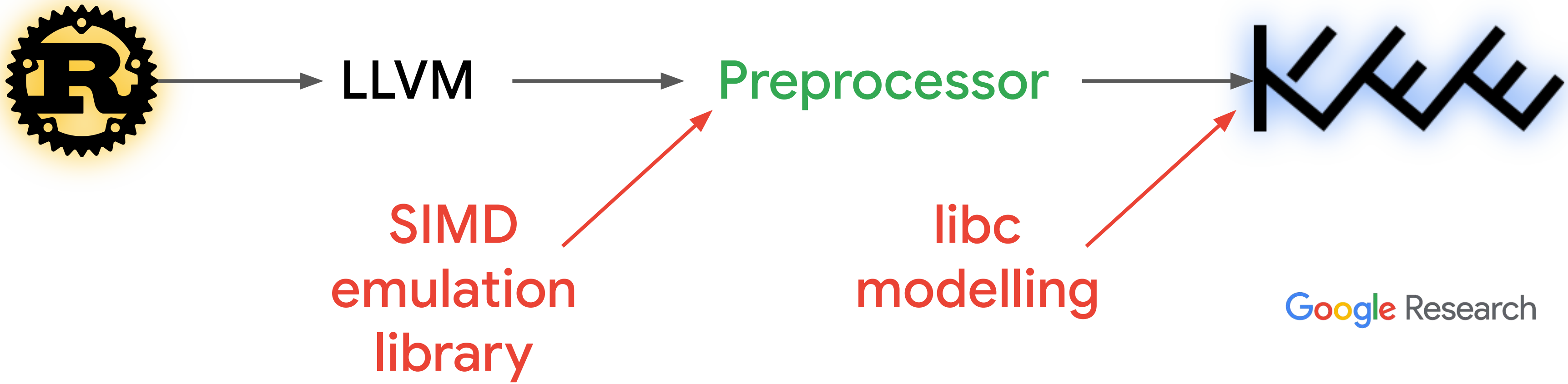
Stdlib features: glibc initializers, libc calls, ...

Popular crates: x86 vector intrinsics

How to add support for Rust features?

Most reusable

1. Extend KLEE
2. LLVM preprocessor
3. Write library / runtime / emulation library
4. Rust compiler flags



The challenge: so many features

Language features: tuples, closures, traits, …

Compiler features: fshl/fshr, LLVM-11, …

Runtime features: ~~concurrency~~

C-Rust interoperation: Rust <-> C foreign calls

Stdlib features: glibc initializers, libc calls, …

Popular crates: x86 vector intrinsics

Key: KLEE, Library, Preprocessor, Rustc flags

Google Research

Experience of using KLEE (still early stages)

Small libraries: base64, memchr, prost (protobufs)

- decode(encode(x)) == x, decode(x) != panic

Applications: uutils/coreutils (96 applications: ls, cat, df, ...)

- Minor bugs found with UTF8
- Have not done a thorough run of all applications

Linux drivers: Rust-for-Linux (demos of how to write LKMs in Rust)

- Challenges: how *not* to run KLEE on entire kernel (mocking), compiling KLEE runtime the right way
- Currently figuring out what a good test harness looks like

Android: keystore2

Summary

Two APIs

- Direct KLEE API
- “Fuzzer” API built on top of KLEE API

Most Rust features supported

- Missing: concurrency, inline assembly

Focus on reusable ways of adding support

- Reusable with other Rust tools. Maybe C/C++ too?

Starting to use on larger codebases

Thank You

Alastair Reid, Shaked Flur
{adreid, sflur}@google.com
@alastair_d_reid, @FlurShaked

More details
https://project-oak.github.io/rust-verification-tools/2021/03/29/klee-status.html

Google Research