Badlock: Static Reentrant Deadlock Detection for Rust via Taint Analysis

Liam Monninger
Department of Computer and Information Science, SEAS
University of Pennsylvania
Philadelphia, PA, USA

***Abstract*—The badlock**

***Index Terms*—static analysis, deadlocks, rust, taint analysis**

I. Introduction

A. Taint Analysis

B. Rust std::sync

II. Badlock

A. Datalog Program

B. Evaluation

We are currently capable of detecting deadlocks in the following simple programs. Programs of this form were defined as the original POC.

1) No Deadlock: We were able to successfully analyzing clearly non-deadlocking Rust programs wherein no lock is ever acquired. The below is an example of a passing program.

```
fn main(){

    let a = 0;
    for i in 0..10 {
        let b = a + i;
        println!("{}", b);
    }

}
```

2) Simple Deadlock: We successfully analyzed simple deadlocking programs which use std::sync::Mutex such as the following.

```
use std::sync::Mutex;

fn main() {

    let mut safe_x = Mutex::new(64);

    let mut guard = safe_x.lock().unwrap();
    *guard += 1;
    println!("Here is fine x: {}", *guard);

    let mut deadlock = safe_x.lock().unwrap();
    *deadlock += 1;
    println!("Should never get here x: {}", *deadlock);

}
```

3) Appropriate Lock Acquisition: We successfully analyzed simple programs where locks are acquired and locked in a manner preventing reentrancy. Notably, we successfully analyzed the implicit lock releases that occur when a Rust lifetime ends.

```
use std::sync::Mutex;

fn main() {

    let mut x = 64;
    let mut safe_x = Mutex::new(x);

    {
        let mut guard = safe_x.lock().unwrap();
        *guard += 1;
        println!("Here is fine x: {}", *guard);
    }

    {
        let mut no_deadlock = safe_x.lock().unwrap
        *no_deadlock += 1;
        println!("Should also get here x: {}", *no_
    }

}
```

III. Conclusion

References