RUSTFUZZ: Scalable Concolic Security Fuzzing Tool for Rust

Mohammadreza Ashouri
ashouri@uni-potsdam.de
University of Potsdam

1 Introduction

Rust is a popular system programming language introduced by Mozilla in 2010 that provides strong compile-time correctness and high performance. The language has improved upon the ideas of other system languages, such as C++. For instance, the Rust compiler provides a reliable memory protection mechanism by performing a robust control over the memory life cycle to eradicate common reliability issues (e.g. memory violation and data race) in system programming languages. Consequently, a wide range of programs, such as operating systems (e.g. RustOS), web frameworks (e.g. Rocket), and blockchain clients (e.g. Parity Ethereum Client) have been built in Rust.

In this work, we have evaluated the security of the Rust compiler in a real-world situation on a new benchmark suite, including core libraries and popular open-source projects. Our analysis aims to measure the usefulness of the memory model and safety structure of the Rust memory model in practice. Particularly, we aim to determine the trade-off between high-level safety guarantees and low-level control over the memory model in this compiler. Accordingly, we implemented a scalable concolic fuzzing tool for the Rust compiler, which is called "RUSTFUZZ", and we have evaluated our approach on *10,693* real-world projects and core libraries written in Rust. As a result, we could successfully identify *19* actual security vulnerabilities in our collected benchmark suite.

Contributions. The main contributions of our work are as follows:

1. We have designed RUSTFUZZ as a practical and scalable security fuzzing tool for performing scalable security analysis in Rust. Our tool works without the source code, based on a parallel concolic testing module. In our analysis, RUSTFUZZ could successfully report multiple security issues, some of which can impose severe security threats to the end-users.
2. We have analyzed the safety of the provided core libraries (e.g., network, cryptography, process and file management) for Rust developers, and we found security issues are rising by using *unsafe blocks* in RUST projects, which results in compromising the type safety in the compiler.

2 Security Analysis

Give the fact that Rust uses LLVM as the backend and produces the LLVM IR as an intermediate outcome of its compilation process; we have implemented our analysis system basedon the LLVM bytecode. Thus, RUSTFUZZ takes the LLVM as the IR code and leverages a distributed concolic execution engine to identify vulnerable execution path and trigger them with concrete values. We illustrated a cycle of a program under analysis in Figure 1. We have also specified a wide range of potential system vulnerabilities associated with memory boundaries issues (e.g. Integers, array access, Stack and Heap) for the root engine of RUSTFUZZ. Figure 2 represents the abstract structure of RUSTFUZZ.

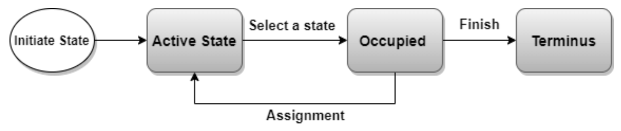

Figure 1: showing the abstract of Condition Life Cycle.

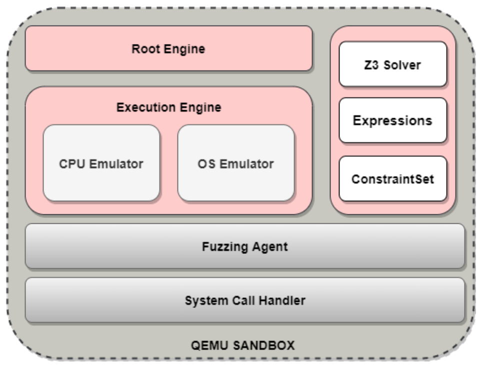

Figure 2: Overview structure of RUSTFUZZ.

We have written RUSTFUZZ in RUST, and we used Ubuntu Linux 16.04 LTS alongside with the QEMU emulator as a fast and portable dynamic translator to provide a cross-platform environment to track any suspicious activities associated with memory issues.

3 Benchmark Collection & Preliminary Result

In order to perform our security analysis, we have collected *10,693* popular open-source Rust project along with various core libraries available at the *Crates.io* repository. Our corpus comprises *14,796* Rust files, which holds *651,193* lines of code (LoC) in total. In Table 1, we represent the summary of our security analysis on some of the benchmarks in our collected corpus.

Table 1: SECURITY ISSUES DETECTED BY RUSTFUZZ

Benchmark	Memory Leak	Denial of Service	Stack Overflow	Heap Overflow	Integer Overflow	Insecure Deserialization	Yaml Injection	Side Channel	Format String
Lucet	✗	✓	✗	✗	✗	✗	✗	✗	✗
Trust-DNS	✗	✓	✓	✗	✗	✗	✗	✗	✗
Rust-Base64	✗	✗	✗	✓	✗	✗	✗	✗	✗
Standard HTTP	✗	✗	✗	✗	✓	✗	✗	✗	✗
Safe-Transmute	✗	✗	✗	✓	✓	✗	✗	✗	✗
Yaml	✗	✗	✗	✗	✗	✗	✓	✗	✗
LibSec	✗	✗	✗	✗	✗	✗	✗	✓	✗
CBOR	✗	✗	✗	✗	✗	✓	✗	✗	✗