RUSTY: A Fuzzing Tool for Rust

Mohammadreza Ashouri
research@ashoury.de
Berlin, Germany

Introduction

Rust is known as one of the most popular programming languages on the Stack Overflow website in 2020, indicating that many programmers have had the opportunity to use Rust in different projects. There are many reasons for this success, mostly due to its performance and safety. Rust is a friendly compiler with useful error messages, which provides excellent documentation with an integrated package manager. Furthermore, it is memory-efficient and fast without a garbage collector. Hence, the Rust compiler can power performance-critical services run on embedded devices. Likewise, due to Rusts rich type system and ownership model, it guarantees memory-safety and thread-safety enabling the end-users to reduce various bug types at compile-time.

However, even though all the advantages come with Rust, we found there are still various security issues in this ecosystem that can compromise the safety of Rust programs. To perform our security analysis, we created RUSTY that perform bug fuzzing based on the combination of concolic testing and property-based testing. is also the first kind of its own for the Rust ecosystem. Our preliminary evaluation collected multiple trendy Rust projects on GitHub, and RUSTY could successfully identify various memory security issues in this benchmark suite.

Contributions:

1. In RUSTY's design, property-based testing allows us to test those specific properties of given code hold for arbitrary inputs. If a failure is found, RUSTY automatically finds the minimal test case to reproduce the problem. In our method, input generation is determined based on value instead of type, making our tool much more adaptable.
2. After identifying errors in the target code, RUSTY creates concrete exploits to build reproducible proof-of-concept (POC) exploits. For implementing the automatic exploit generation unit, RUSTY leverages a concolic execution engine based on the Z3 SMT solver [1].

Fuzzing Approach

Fuzzing and property testing include generating random inputs and then verifying if a program fails on those inputs. Fuzzing is commonly a black-box method, meaning we do not attempt to inform it too much on how to go about making these inputs. Randomized property testing, on the other hand, requires extensive familiarity with the code under test. More precisely, it works by testing an executable predicate (a property) on a stream of randomly generated inputs [3, 4]. Hence, manual specification is required both the properties to test and define a rough "form" of inputs that are "unusual". The main benefit of property testing is that it works faster and more precisejust like unit tests.

However, some properties are conditioned on the input satisfying demanding semantic invariants that are not the results of its syntactic structure. E.g., an input list must be sorted or have no duplicates. Most randomly generated inputs fail to satisfy properties with such inadequate preconditions and so are discarded. Consequently, much of the target system may go untested.

In this work, we address this issue with our novel bug fuzzing system called "RUSTY". Our system indeed is a coverage guided, property-based testing fuzzing tool inspired by the related area of coverage guided fuzzing [5], exemplified by tools like AFL [2]. Thus, instead of generating a fresh random input at each iteration, RUSTY can also produce new inputs by mutating previous ones using type-aware, generic mutation operators. The target program is instrumented to track which control flow branches are executed during a run, and inputs whose runs expand control-flow coverage are preserved for future mutations.

This means that, when light conditions in the target program are satisfied, and new coverage is observed, the input that triggered them will be maintained and used as a springboard to go further. We have implemented RUSTY and a concolic execution engine (based on Z3 SMT solver) to produce concrete exploit values for discovered bugs for rust programs. Figure 1 show the overview architecture of RUSTY.

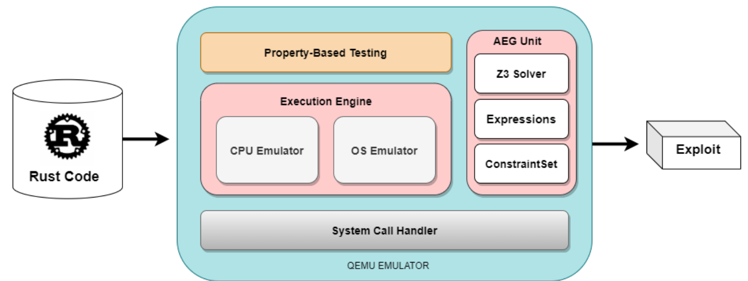

Figure 1: Overview structure of RUSTY.

Preliminary Evaluation

To evaluate the effectiveness of RUSTY, we tested it on Rust's vulnerable libraries. Table 1 shows some of the identified vulnerabilities by fuzzing tool in our benchmark suite.

Table 1: Vulnerabilities detected by RUSTY

Detected Vulnerability	Memory Leak	Denial of Service	Buffer Overflow	Heap Overflow	Integer Overflow	Insecure Deserialization	Code Execution	Side Channel	Format String
CVE-2017-1000430	✗	✗	✓	✗	✗	✗	✗	✗	✗
CVE-2018-1000622	✗	✗	✗	✗	✗	✗	✓	✗	✗
CVE-2018-1000810	✗	✗	✓	✓	✗	✗	✗	✗	✗
CVE-2019-1010299	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2019-15544	✗	✓	✗	✗	✗	✗	✗	✗	✗
CVE-2020-25016	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2020-25791	✗	✗	✓	✗	✗	✗	✗	✗	✗
CVE-2020-25792	✗	✗	✓	✗	✗	✗	✗	✗	✗
CVE-2020-25793	✗	✗	✓	✗	✗	✗	✗	✗	✗

References

[1] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[2] Peter Gutmann. Fuzzing code with afl. *login:*, 41(2), 2016.

[3] Leonidas Lampropoulos, Michael Hicks, and Benjamin C Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[4] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 398–401, 2019.

[5] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. Safl: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 61–64, 2018.