SymRustC: A Hybrid Fuzzer for Rust

Frédéric Tuong
Simon Fraser University
Canada

Mohammad Omidvar Tehrani
Simon Fraser University
Canada

Marco Gaboardi
Boston University
USA

Steven Y. Ko
Simon Fraser University
Canada

ABSTRACT

We present SymRustC, a hybrid fuzzer for Rust. SymRustC is hybrid in the sense that it combines fuzzing and concolic execution. SymRustC leverages an existing tool called SymCC for its concolic execution capability and another existing tool called LibAFL for its fuzzing capability. Since SymCC instruments LLVM IR (Intermediate Representation) for concolic execution and the Rust compiler uses LLVM as a backend, we integrate SymCC with the Rust compiler to instrument Rust programs for concolic execution. LibAFL provides a framework to develop a fuzzer, and we use it to develop a hybrid fuzzer that combines fuzzing and our concolic execution. We discuss our implementation as well as four case studies to demonstrate that SymRustC can generate inputs that discover errors in Rust programs.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**; **Software testing and debugging**.

KEYWORDS

Hybrid fuzzing, concolic execution, Rust.

ACM Reference Format:
Frédéric Tuong, Mohammad Omidvar Tehrani, Marco Gaboardi, and Steven Y. Ko. 2023. SymRustC: A Hybrid Fuzzer for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604927

1 INTRODUCTION

Fuzzing is a popular technique for automated test case generation. It is based on the simple idea of input mutation that modifies already-tested inputs to generate new inputs to test. There are many tools, libraries, and services publicly available [4–8, 14] and developers increasingly consider fuzzing an essential tool for testing.

Hybrid fuzzing [10, 12], on the other hand, combines fuzzing with another automated test case generation technique called concolic execution. Concolic execution gathers path constraints from the program being tested and uses an SMT solver [3] to generate inputs that satisfy those path constraints. While fuzzing is good at quickly running a program repeatedly with new test cases, it has a well-known limitation that it does not explore code paths with tight branch conditions. Hybrid fuzzing aims to address this problem by supplementing fuzzing with concolic execution that is good at exploring code paths with tight branch conditions.

This paper presents SymRustC, a hybrid fuzzer for Rust programs. Though there are many fuzzing and other types of automated test case generation tools available for Rust [6, 9], we see that there are no hybrid fuzzing tools currently available. To fill this gap, we have developed SymRustC. Since Rust is rapidly becoming a language of choice for many developers who desire memory safety without sacrificing performance, we hypothesize that a hybrid fuzzing tool can help make Rust programs even more reliable.

Our goal is to rapidly develop a usable hybrid fuzzing tool and hence we leverage existing tools as much as possible to accomplish our goal. More specifically, we use two existing tools as building blocks to develop SymRustC. The first building block is SymCC [11], a compilation-based concolic execution tool for C and C++ programs. SymCC instruments a program at the LLVM IR (Intermediate Representation) level to enable concolic execution for the program. Since the Rust compiler uses LLVM as compiler backend, it gives us an opportunity to integrate SymCC with the Rust compiler to instrument Rust programs for concolic execution.

The second building block is LibAFL [5], which is a framework that allows fuzzer developers to write a custom fuzzer. It provides pluggable and replaceable fuzzer components, such as an input mutator component, an input corpus component, a progress monitor component, etc., where developers can write their own algorithms for these components or use built-in ones provided by LibAFL. Using this, we develop SymRustC that combines fuzzing and concolic execution with our SymCC-enabled Rust compiler. We discuss how we use these building blocks further in Sec. 3.2.

For evaluation, we present four case studies where we use SymRustC to find bugs discovered by an existing tool, `cargo-fuzz`. This is to demonstrate SymRustC's baseline that it is able to find bugs in real Rust programs. We discuss the case studies further in Sec. 4.

SymRustC is available at: https://github.com/sfu-rsl/symrustc. In addition, our concolic execution engine (used as part of our hybrid fuzzing) is available separately at: https://github.com/sfu-rsl/symrustc_toolchain.

2 RELATED WORK

SymRustC's concolic execution engine is based on SymCC [11], but targets Rust programs rather than C and C++ programs.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3604927

SymCC is a compilation-based concolic execution tool for C and C++ programs. It instruments a program to perform concolic execution at compile time. In order to be effective, it needs to compile not only the program being tested but the dependencies as well, e.g., the standard library of C and C++. SymCC demonstrates the benefit of using a compilation-based concolic execution approach, which is the speed gain on concolic-execution at run time. Compared to other approaches, such as QSYM [13] that performs run-time instrumentation or KLEE [1] that dynamically interprets LLVM bitcode, SymCC is shown to be faster by the average factors of 10 to 12.

Concolic execution requires program libraries to be all instrumented as well. However, there are other research directions analyzing the trade-off of static vs. dynamic compilation of all libraries, and exploring different instrumentation strategies: see for instance the hybrid solution proposed by SymFusion [2].

As we describe further in Sec. 3.2, we use LibAFL [5] to combine concolic execution and fuzzing. LibAFL already has initial integration with SymCC to support hybrid fuzzing for C and C++ programs. We use this support to further enable hybrid fuzzing for Rust programs.

For concolic execution or hybrid fuzzing, LibAFL uses Z3 [3] as its SMT solver. Z3 is also used by SymCC. Thus, SymRustC also uses Z3 as SMT solver.

3 HYBRID FUZZING WITH SymRustC

SymRustC combines concolic execution and fuzzing to perform hybrid fuzzing for Rust programs. Its fuzzing generates new inputs by input mutation. Its concolic execution generates new inputs by solving SMT queries constructed from the path constraints of the program being tested. SymRustC uses all of these inputs to repeatedly test the program. The main capabilities of concolic execution and fuzzing come from SymCC and LibAFL, which we integrate together as we describe further below.

3.1 Concolic Execution

SymRustC's concolic execution engine can be viewed as a compiler: starting from a Rust source program as an input, the tool produces an *instrumented* compiled binary that runs in a concolic fashion. To do so, the instrumentation keeps track of the program state during execution, gathers path constraints, and sends queries to an SMT solver to generate new inputs.

We implement SymRustC's concolic execution engine as a modification of the Rust compiler (`rustc`) that uses SymCC as a compiler backend. This is to take advantage of the fact that the Rust compiler uses LLVM as default backend, and that SymCC is essentially an LLVM-pass performing instrumentation for concolic execution on LLVM's Intermediate Representation (IR).

One might think that the simplest way to enable such a SymCC-enabled Rust compiler is to add the SymCC pass to LLVM and compile it together with the rest of the Rust compiler source to produce a new compiler. In fact, the Rust compiler allows us to control which LLVM passes to include during a compile process (using the option `-C 'passes=P'`). Thus, one could imagine using this option to enable the SymCC pass as needed.

While this may seem like the easiest approach, we have found out that there is one critical limitation. A Rust program almost always uses the Rust standard library, which is pre-compiled and only linked (not compiled) during the program's compile process. What this means is that the Rust standard library will not perform concolic execution even when we enable the SymCC pass during the compile process of a Rust program.

To address this problem, we modify the compile process of the Rust compiler itself. This is mainly because we determined that it would be the most natural way—the standard library gets compiled together when compiling the Rust compiler. Thus, we first enable the SymCC pass for LLVM and then use the SymCC-enabled Rust compiler to compile the standard library, all during the compile process of the Rust compiler.

Briefly, the Rust compiler is divided in two parts: `src/bootstrap` configuring how the Rust source in Rust should be compiled (e.g. which `-C passes` options are enabled) and `compiler` containing the core implementation of the Rust compiler. Our modifications are only made in `src/bootstrap` in about 20 lines of code in addition to adding a modified LLVM with the SymCC pass and pointing the Rust compile process to use it.

3.2 Hybrid Fuzzing with LibAFL

To implement a hybrid fuzzer using our concolic execution engine, we use LibAFL [3], which is a modular fuzzing framework written in Rust. LibAFL breaks the fuzzing process into several components and allows developers to customize them or replace them with better algorithms. These components include, (i) Corpus, which holds test cases for fuzzing, (ii) Observer and Feedback, which check and determine whether an input is interesting to be included in the corpus, (iii) Stage, which implements a strategy to process each test case in the corpus, and (iv) Executor, which is responsible for running the target program against an input. For example, to perform fuzzing for a function in a library, one can have an Executor that is simply a testing harness calling the desired function with the current input, connect it with an Observer and Feedback that monitor and report block coverage, initialize a Corpus with a seed test case, employ a Stage that uses random mutation to generate the new inputs, and finally run the fuzzing loop.

An advantage of using LibAFL for us is that LibAFL has built-in support for hybrid fuzzing with SymCC. Since our concolic execution is essentially done by SymCC, we can leverage LibAFL's hybrid fuzzing support as well. More specifically, LibAFL provides a specialized type of Stage that runs an executable compiled by SymCC, generates new test cases based on the path constraints and the corresponding SMT solver results from the concolic execution, and evaluates those new test cases for further fuzzing. To combine such concolic execution with pure fuzzing, LibAFL provides a *client-server* mechanism in which multiple clients perform either concolic execution or pure fuzzing, and the server maintains a shared state to prioritize test cases based on metrics such as edge coverage. Our implementation creates two clients, one for pure fuzzing and the other for concolic execution, though we could create more than one client to further speed up.

3.3 Limitations

Although SymRustC enables concolic execution and hybrid fuzzing for Rust programs, it relies on existing tools and inherits the same

limitations from them. The most notable one is that SymCC does not support all LLVM instructions, which affects the precision of our concolic execution. Since we do not control which LLVM instructions that the Rust compiler generates, it is difficult to limit the scope of this problem. However, SymCC supports a sufficient subset of LLVM instructions for our experiments as we show in Sec. 4.

4 EVALUATION

We demonstrate how SymRustC performs in comparison to two other fuzzers—(i) an existing fuzzer for Rust, `cargo-fuzz` [6], and (ii) a custom fuzzer that we implement using `LibAFL` named `LibAFL Fuzz`. `cargo-fuzz` is a popular fuzzer for Rust that is integrated with Rust's build system, `cargo`. It is a wrapper for Rust programs that uses an existing fuzzer underneath, and has support for multiple fuzzer targets. We use `libFuzzer` as the target fuzzer. `LibAFL Fuzz` uses a mutation strategy called `HavocMutator` provided by `LibAFL` and uses code coverage provided by LLVM's SanitizerCoverage instrumentation[1]. We discuss four case studies where all tools are able to find a common problem. The purpose is to show the baseline performance of SymRustC that it can find problems (albeit known) in real Rust programs. We select the four cases from the Rust Fuzzing Authority's Trophy Case repository[2] that is a collection of problems found in Rust programs mostly using `cargo-fuzz`. Except for one case, all of the problems are already resolved in the newer versions of the corresponding programs.

4.1 csscolorparser

Program Description: `csscolorparser`[3] is a library for parsing color strings defined in W3C's CSS Color Module Level 4. It supports a variety of formats including hexadecimal, named colors, and color functions such as `rgb` and `hsv`.

Bug Description: While the `parse` function is expected to be robust and indicate any invalid input by an `Err` object, a string containing non-ASCII characters can cause a panic. For example, the following code triggers the problem:

```
csscolorparser::parse("\u{023A}");
```

Result: As almost any string containing a non-ASCII character can trigger the bug, the tools are expected to hit it quickly. In fact, `cargo-fuzz` was able to find it on the first try. It took 5 executions for the hybrid fuzzer to find it, whereas the pure fuzzing client took 45 executions. All experiments took less than a second.

4.2 mp4ameta

Program Description: `mp4ameta`[4] is a Rust library that reads and writes iTunes-style MPEG-4 audio metadata. According to the Rust package registry `crates.io`, it has a download count of 46,881 as of May, 2023.

Bug Description: A failure occurs while executing the following piece of code:

```
let mut data = std::io::Cursor::new(data);
mp4ameta::Tag::read_from(&mut data);
```

with the following input:

```
00 00 00 01  66 74 79 70  00 84 FF FF  FF FF 00 84
```

The execution produces the following error message:

```
memory allocation of 37436171902451828 bytes failed
Aborted (core dumped)
```

At the time of writing, the origin of this error is still in investigation by the project authors[5]. However, instead of abnormally exiting the program, we would expect `mp4ameta::Tag::read_from` to not fail but return a result that indicates an error.

Result: Running `cargo-fuzz` finds the error quickly in less than 1 second after at least 27 executions. Similarly, it takes less than 1 second to find the error for SymRustC in 94 executions. After running the experiment for 5 minutes, we have observed that the number of errors gets continuously increased over time, reaching a total of 1919 at the end of our experiment. Note that these errors are exclusively reported by the concolic client: the fuzzing client was not able to find a problem during the experiment. Even when running in standalone mode (still in 5 minutes), the fuzzing client was not able to find an error (after at least 311K executions).

4.3 libflate

Program Description: `libflate`[6] implements the Deflate algorithm to decode a Gzip stream. The library is popular as it has more than 7 million downloads as of May 2023.

Bug Description: The following piece of Rust code shows how one can use the library to decode an encoded input:

```
let mut decoder = Decoder::new(&input[..]);
std::io::copy(&mut decoder, &mut std::io::sink());
```

However it panics with the following input:

```
05 05 00 05  01 01 00 00  05 5B 62
```

Result: Running `cargo-fuzz` finds the error quickly in less than 2 seconds after at least 8815 executions. Similarly, it takes less than 2 seconds to find the error for SymRustC in 937 executions. After running the experiment for 5 minutes, we have observed that the number of errors gets continuously increased over time, totaling 241 at the end of our experiment. In contrast to `mp4ameta`, these errors are exclusively reported by the fuzzing client this time: the concolic client did not find any problems during the experiment. When run in standalone mode, we have observed a similar behavior: the fuzzing client found the first error in 2 seconds in 917 executions.

4.4 bincode

Program Description: `bincode`[7] is a binary encoder and decoder library operating on byte arrays or streams. It facilitates encoding for custom types through attributes, and provides default encoders and decoders for a wide range of types in the core and standard libraries. As of May 2023, it is downloaded over 32 million times through `crates.io`, and over 2 thousand packages depend on it.

Bug Description: There is an unhandled error (in `2.0.0-beta.0`[8]) when decoding a byte array holding the maximum values for two unsigned integers as a `std::time::Duration` like below.

[1] https://clang.llvm.org/docs/SanitizerCoverage.html
[2] https://github.com/rust-fuzz/trophy-case
[3] https://crates.io/crates/csscolorparser
[4] https://crates.io/crates/mp4ameta
[5] https://github.com/Saecki/mp4ameta/issues/25
[6] https://crates.io/crates/libflate
[7] https://crates.io/crates/bincode
[8] https://github.com/bincode-org/bincode/pull/465

Table 1: Number of random re-tries within the given time until potentially finding a bug, using cargo-fuzz (cargo F), LibAFL in pure fuzzing mode without concolic clients (LibAFL F), and SymRustC.

Program	Execution re-tries			Time			Bug found?		
	cargo F	LibAFL F	SymRustC	cargo F	LibAFL F	SymRustC	cargo F	LibAFL F	SymRustC
csscolorparser	1	45	5	<1 sec	<1 sec	<1 sec	yes	yes	yes
mp4ameta	27	311 K	94	<1 sec	300 sec	<1 sec	yes	no	yes
libflate	8815	917	937	2 sec	2 sec	2 sec	yes	yes	yes
bincode	1519	97	66 K	<1 sec	<1 sec	58 sec	yes	yes	yes

```
let mut buff = [0u8; 14];
let nums = (u64::MAX, u32::MAX);
bincode::encode_into_slice(&nums, &mut buff, config);
let result: Result<(Duration, usize), _> =
  bincode::decode_from_slice(&mut buff, config);
```

The failure occurs in the default decoder provided for `Duration` where it passes the decoded values for `secs` and `nanos` directly to the constructor and consequently inherits the panic raised by it based on the overflow check.

```
let secs = Decode::decode(&mut decoder)?;
let nanos = Decode::decode(&mut decoder)?;
Ok(Duration::new(secs, nanos))

pub const fn new(secs: u64, nanos: u32) -> Duration {
  let add_secs = (nanos / NANOS_PER_SEC) as u64;
  let secs = match secs.checked_add(add_secs) {
    None => panic!("overflow in Duration::new"),
// ...
```

Result: Using `cargo-fuzz`, the bug was hit in approximately 1519 executions. For `SymRustC`, it took around 66K executions to hit it for the first time. Additionally, running the fuzzing client alone led to finding the error in about 97 executions. We hypothesize that the difference may come from the lack of support for the overflowing binary operations in `SymCC` (as of January 2023).

5 CONCLUSION

In this study, we have explored leveraging `LibAFL` and `SymCC` to develop `SymRustC`, a hybrid fuzzer for Rust. While `SymCC` targets LLVM-IR and is supposed to be compatible with Rust, we have found that integrating it into the Rust compiler and instrumenting the standard library are among the challenges of the development. Additionally, the lack of support for some LLVM instructions in `SymCC` is a limitation for `SymRustC`.

Nonetheless, our results show the obtained concolic execution is sufficient to test real Rust program with hybrid fuzzing. In the four cases we discuss in this paper, `SymRustC` is able to generate test cases that trigger bugs. For our future work, we can improve the instrumentation process for code coverage and concolic execution. Currently, they are done in two separate steps, but we believe that we can combine them together in a single step.

ACKNOWLEDGMENTS

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Amazon Research Awards.

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[2] Emilio Coppa, Heng Yin, and Camil Demetrescu. 2022. SymFusion: Hybrid Instrumentation for Concolic Execution. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 100:1–100:12. https://doi.org/10.1145/3551349.3556928

[3] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[4] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[5] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 1051–1065. https://doi.org/10.1145/3548606.3560602

[6] github.com 2023. cargo-fuzz. Retrieved May 18, 2023 from https://github.com/rust-fuzz/cargo-fuzz

[7] github.com 2023. Honggfuzz. Retrieved May 18, 2023 from https://github.com/google/honggfuzz

[8] github.com 2023. OSS-Fuzz. Retrieved May 18, 2023 from https://github.com/google/oss-fuzz

[9] github.com 2023. Rust Verification Tools. Retrieved May 18, 2023 from https://github.com/project-oak/rust-verification-tools/

[10] Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 416–426. https://doi.org/10.1109/ICSE.2007.41

[11] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[12] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[13] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[14] Michał Zalewski. 2023. American Fuzzy Lop (AFL). Retrieved May 18, 2023 from https://lcamtuf.coredump.cx/afl/

Received 2023-05-18; accepted 2023-06-08