CrabSandwich: Fuzzing Rust with Rust (Registered Report)

Addison Crump
CISPA, Germany
addison.crump@cispa.de

Dongjia Zhang
EURECOM, France
zhang@eurecom.fr

Syeda Mahnur Asif
CISPA, Germany
syeda.asif@cispa.de

Dominik Maier
TU Berlin, Germany
dmaier@sect.tu-berlin.de

Andrea Fioraldi
EURECOM, France
fioraldi@eurecom.fr

Thorsten Holz
CISPA, Germany
holz@cispa.de

Davide Balzarotti
EURECOM, France
balzarot@eurecom.fr

ABSTRACT

The Rust programming language is one of the fastest-growing programming languages, thanks to its unique blend of high performance execution and memory safety. Still, programs implemented in Rust can contain critical bugs. Apart from logic bugs and crashes, code in *unsafe* blocks can still trigger memory corruptions. To find these, the community uses traditional fuzzers like LibFuzzer or AFL++, in combination with Rust-specific macros. Of course, the fuzzers themselves are still written in memory-unsafe languages.

In this paper, we explore the possibility of replacing the input generators with Rust, while staying compatible to existing harnesses. Based on the Rust fuzzer library LibAFL, we develop CrabSandwich, a drop-in replacement for the C++ component of cargo-fuzz. We evaluate our tool, written in Rust, against the original fuzzer LibFuzzer. We show that we are not only able to successfully fuzz all three targets we tested with CrabSandwich, but outperform *cargo-fuzz* in bug coverage. During our preliminary evaluation, we already manage to uncover new bugs in the *pdf* crate that could not be found by cargo-fuzz, proving the real-world applicability of our approach, and giving us high hopes for the planned follow-up evaluations.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*.

KEYWORDS

fuzzing; framework

ACM Reference Format:
Addison Crump, Dongjia Zhang, Syeda Mahnur Asif, Dominik Maier, Andrea Fioraldi, Thorsten Holz, and Davide Balzarotti. 2023. CrabSandwich: Fuzzing Rust with Rust (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop (FUZZING '23), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605157.3605176

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FUZZING '23, July 17, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0247-1/23/07...$15.00
https://doi.org/10.1145/3605157.3605176

1 INTRODUCTION

In recent years, the Rust programming language has emerged as a compelling choice for developing secure and high-performance software systems. Rust's unique design philosophy, combining strong memory safety guarantees with an expressive and modern syntax, offers developers a powerful tool to mitigate common programming pitfalls such as memory leaks, null pointer dereferences, and data races. The language's emphasis on zero-cost abstractions and minimal runtime overhead has made it particularly attractive for systems programming, network services, and embedded applications.

However, despite Rust's built-in safety features, software faults, and security vulnerabilities can still surface due to complex interactions, corner cases, and unforeseen scenarios. Denial-of-service is not the only kind of vulnerability concerning Rust code but the usage of `unsafe` code such as foreign function interface (FFI) invocations to memory unsafe languages with broken assumptions and invariants can propagate these issues to the "safe" part of the codebase. Traditional testing methodologies, such as unit testing and manual code review, have limitations in detecting these subtle flaws and ensuring comprehensive coverage of the codebase.

This is where fuzz testing (*fuzzing* for short) plays a crucial role in identifying software vulnerabilities in Rust. Fuzzing complements traditional testing methods by automatically generating and injecting a variety of random or targeted inputs into a program, systematically exploring its execution paths, and identifying potential weaknesses. By subjecting Rust programs to unexpected and unconventional inputs, fuzz testing can uncover hidden bugs, memory corruption issues, undefined behaviors, and other security vulnerabilities that might otherwise remain undetected.

The standard tools to deploy fuzzing on Rust codebases are wrappers around popular fuzzers for C/C++ targeting LLVM bitcode for instrumentation. One of the most used tools in practice is cargo-fuzz, a wrapper around LLVM's LibFuzzer [28], both because Rust is built on LLVM and is thus trivially compatible, and because LibFuzzer proves to be an effective in-process fuzzer. With the importance of fuzzing for ensuring safety in Rust programs, it is critical to ensure that developers and researchers have the most effective fuzzing strategies available. Unfortunately, LibFuzzer recently entered maintenance mode [30] and will thus only receive bug fixes in the future. In effect, LibFuzzer's development has stopped.

In this paper, we present CrabSandwich, a cargo-fuzz-compatible fuzzer runtime using LibAFL [18] – a new powerful framework for fuzzers development in Rust– instead of LibFuzzer designed to address the challenges of testing Rust programs. We discuss the key features and design principles of CrabSandwich, showcasing its ability to efficiently generate diverse inputs and provide insightful feedback to aid vulnerability discovery. In short, our LibFuzzer replacement allows changing nothing in cargo-fuzz while maintaining its widely used interface for fuzzing the same, while we implement a LibFuzzer replacement with many advanced features integrated into LibAFL such as the corpus scheduler from AFL++ [17] or the Grimoire [10] mutator for textual inputs.

In this preliminary report, we showcase a preliminary small-scale experiment involving just three target programs. CrabSandwich was able to find two more bugs than cargo-fuzz in one of those targets, bugs that are under the process of being reported to the project authors at the time of writing, and our prototype was able to reach more code coverage than the baseline tool in two out of three benchmarks. We hope that the preliminary evaluation can hint that CrabSandwich is an improvement for the fuzzing ecosystem in Rust and, through comprehensive evaluation and case studies, in the complete evaluation, we will demonstrate the effectiveness of CrabSandwich in detecting real-world vulnerabilities in a significant amount of Rust projects, contributing to the community with a dataset of applications that can be reused to evaluate future fuzzers targeting Rust programs.

As a final goal, we aim to provide an up-to-date alternative to LibFuzzer using the latest advancements in fuzzing research and with an active community maintaining it to the Rust ecosystem without any effort for the end user that has only to change a dependency in the cargo configuration file of the target program under test. To achieve that, every artifact will be free and open source software and the CrabSandwich implementation will be available on `crates.io` to the end users to facilitate adoption.

Contributions. In short, we make the following contributions in this paper:

- We develop and open-source CrabSandwich, a cargo-fuzz-compatible fuzzer runtime utilizing LibAFL for its implementations of recent fuzzing advancements
- We create a benchmark suite of open-source Rust projects to evaluate CrabSandwich, cargo-fuzz, and future Rust fuzzers
- We propose a range of improvements to and advancements in LibAFL to improve compatibility and performance for Rust targets
- We develop and improve existing Rust target harnesses, responsibly disclosing discovered bugs and contributing test harnesses to the project developers
- We contribute missing pieces to cargo-fuzz and `libfuzzer-sys` to expand compatibility for other runtimes, including CrabSandwich and others.

2 BACKGROUND

In this section, we discuss the background concepts on which our approach is built, namely fuzz testing, Rust code testing, and the LibAFL fuzzing framework.

2.1 Fuzz Testing

Fuzz testing, also known as fuzzing, is a widely-used technique for discovering software vulnerabilities. It involves running a target program with mutated inputs in quick succession, in order to trigger novel and potentially buggy program points. The first fuzzers appeared in the early 1990s [37] and relied primarily on *blackbox* testing, which involves providing the program under test with randomly generated inputs and using crashes and error conditions as the only guidelines for the fuzzing campaign. More advanced blackbox fuzzers [14] take into account the structure information of test cases for their mutations, but the limitations of such approaches can make them ineffective at bypassing even simple conditional statements.

To overcome the lack of target introspection in blackbox fuzzing, two other paradigms exist: *whitebox* and *graybox* fuzzing. Whitebox fuzzing [20] relies on complex instrumentations and code analysis to inspect the state space of the target systematically, but introduces a non-negligible performance slowdown. Graybox fuzzing [17, 28, 45], on the other hand, uses only lightweight code instrumentation – usually to trace code coverage – to produce feedback that is used to evaluate the quality of a test case that is kept for further mutations if "interesting". This approach has become the leading technique to discover vulnerabilities in modern codebases, thanks to the popularity of AFL [19, 51] and Google's OSS-Fuzz [1] being a prime example of a large-scale deployment of graybox fuzzing.

Researchers continue to refine graybox fuzzing techniques by developing new methodologies for mutating test cases [3, 21, 33, 39], managing test cases [11, 43, 48], providing feedback on program behavior [15, 23, 35, 47, 49] and more [13, 34, 41, 50].

2.2 Rust and the Rust Ecosystem

The Rust programming language is a programming language which features memory safety, strong typing, and a very powerful build system at its core. For these reasons, Rust has recently exploded in popularity for its ease of use, the simplicity of dependency management, and the guarantees provided by the compiler.

Implementing the paradigm of ownership, the Rust memory model strongly prevents developers from implementing code which causes data races or memory corruption. Users can circumvent parts of these restrictions by the use of the `unsafe` keyword, which indicates that unsafe methods or operations, such as pointer dereferences, may be used, but the safety of the operation must be checked by the user and cannot be guaranteed by the compiler. These regions are notoriously difficult to verify, often having edge cases which may lead to the use of uninitialized memory or indexing outside of valid bounds.

An overwhelming majority of Rust executables and libraries are developed in a "crate", a single organizational unit that can be built, tested, and run with the `cargo` command. Users can specify dependencies for their crates using a markdown language, which will be fetched from the `crates.io` repository or elsewhere, as specified. This dynamic makes dependency and build management simple, and is collectively referred to as the "Rust Ecosystem". In addition, users can install extensions to `cargo`, allowing them to do more complex tasks which involve the crate.

2.3 Testing Rust Code

Albeit being a safe programming language, Rust projects are widely tested with fuzz testing tools to catch panic errors or memory safety issues in unsafe code. The OSS-Fuzz [1] platform continuously tests Rust libraries using cargo-fuzz [6], a cargo– the most used Rust build system – extension that builds the target program and fuzz it with LibFuzzer. Other wrappers around the most used fuzzers for C/C++ code are also available in the Rust fuzzing ecosystem, namely afl.rs [5] that bridge AFL++ to Rust and honggfuzz-rs [7] that does the same for HonggFuzz.

The interface used by cargo-fuzz resembles the harness function that LibFuzzer requires to fuzz C/C++ code. The `libfuzzer-sys` crate exposes the `fuzz_target` macro that takes a harness closure as an argument. The harness closure is called by the fuzzer passing the generated bytes slice as input to the closure. An example is the code in Listing 10, a harness to fuzz Rust's `url` crate.

```
#![no_main]
#[macro_use] extern crate libfuzzer_sys;
extern crate url;

fuzz_target!(|data: &[u8]| {
    if let Ok(s) = std::str::from_utf8(data) {
        let _ = url::Url::parse(s);
    }
});
```

Listing 1: cargo-fuzz harness to fuzz the url crate.

Being an extension of the build system, cargo-fuzz takes care of building the target with the right instrumentation flags for SanitizerCoverage [26], more specifically the inline 8-bit counters instrumentation used by LibFuzzer to track edge coverage at runtime and comparison tracing to identify what values determine the "choice" of conditional branches (such as `if`, `while`, and so on). Additionally, it can compile the target with the LLVM sanitizers to spot silent memory unsafety issues related to unsafe code or C/C++ libraries linked to the Rust code as well as more specialized memory safety sanitizers related to Rust-specific memory model violations.

When a cargo-fuzz fuzzer is executed, the fuzzer harness is compiled, linked to libfuzzer, and then executed by translating cargo-fuzz flags into LibFuzzer flags. From this point onward, the fuzzer behaves identically to a classical LibFuzzer-based C or C++ fuzzer.

2.4 LibAFL

While state-of-the-art fuzzers like AFL++ [17] are effective in finding bugs, their design does not prioritize extensibility, resulting in multiple incompatible fuzzers. A novel fuzzing framework written in Rust, LibAFL [18], solves this problem by providing reusable pieces and a modular design to build custom-tailored fuzzers. It integrates techniques from over 20 previous works providing a solid base for comparative and extensible research.

LibAFL has been designed with three key principles in mind: *extensibility*, *portability*, and *scalability*. It enables the seamless combination of orthogonal techniques and simplifies the development of new components, making it highly extensible. Furthermore, it supports multiple operating systems, including Linux, Windows, macOS, Android, and *BSD, while its core library is system-independent, enabling users to write code that can run even on bare-metal systems. Finally, LibAFL is highly scalable, with an event-based interface for fuzzers communication and an implementation based on shared memory that allows in-process fuzzers to scale linearly over cores.

Core Concepts. The core concepts on which LibAFL is based are these generic components that can be used to describe various fuzzers:

Input: The data taken from external sources that affect the behavior of a program. Different fuzzers may use different representations of inputs, such as byte arrays, system call sequences, abstract syntax trees, or encoded token sequences.

Corpus: It is a storage for inputs and their associated metadata. The corpus can be stored in memory or on disk, each with its own trade-offs in terms of speed and resource usage.

Scheduler: The scheduler selects the next testcase to be fuzzed from the corpus. It can use different policies like FIFO, random selection, or more advanced techniques based on statistics or interesting properties of testcases.

Stage: A stage defines an action performed on a single testcase from the corpus. It can involve mutators that modify the input, analysis stages, or minimization phases to reduce the size of testcases while maintaining coverage.

Observer: An observer provides information about a single execution of the target program. Common observers include coverage maps that track executed edges or other metrics beyond code coverage.

Executor: The executor is responsible for executing the target system using an input from the fuzzer. It defines how the target is executed and handles volatile operations related to each run, such as informing the program about the input.

Feedback: The feedback entity classifies the outcome of a program execution as interesting or not. It uses information from observers to determine if an execution is novel or satisfies specific objectives, such as crashing the program.

Mutator: A mutator generates new derived inputs based on one or more existing inputs. It can perform bit-level mutations or modify the internal representation of inputs, depending on the specific fuzzer and input type.

Generator: A generator is responsible for creating new inputs from scratch. While less common in feedback-driven fuzzing, some fuzzers use generators to create initial inputs, such as random generators or grammar-based generators.

Implementing new components based on this classification is how LibAFL can be extended. To discuss CrabSandwich, we will point to these components discussing what we implemented or changed in our approach.

3 DESIGN

In the following, we discuss the design decisions for CrabSandwich, as depicted in Figure 1. Our main goal was to minimize the user effort when creating new fuzzers and when migrating existing fuzzers to CrabSandwich. For these reasons, we make the user interface for CrabSandwich as similar to cargo-fuzz as possible. A typical workflow for cargo-fuzz is the following:

(1) Install cargo-fuzz with `cargo install -f cargo-fuzz`
(2) Create the fuzz environment with `cargo fuzz init`, generating a new crate containing the fuzz harness(es)

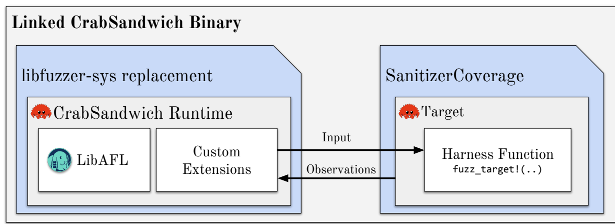

Figure 1: High-level Ovewiew of CrabSandwich: The runtime gets linked into the final binary, and the target is instrumented with the usual LLVM instrumentation passes.

(3) Fill in the fuzz harness with code and dependencies to call and tests the target using fuzzer-provided input.

Internally, cargo-fuzz uses the `libfuzzer-sys` crate to expose the `fuzz_target` macro. This crate additionally compiles and links a recent version of LibFuzzer. Since we intend to keep the workflow unchanged, where possible, we replace the LibFuzzer library linking step to use our library instead. This is the least invasive approach to introduce CrabSandwich. We provide a runtime that keeps the same parameters as LibFuzzer and thus remains compatible with cargo-fuzz. By exporting the symbols defined by `libfuzzer-sys`, we ensure that users do not need to change the harnesses whatsoever and can continue to use their cargo-fuzz infrastructure to benefit from CrabSandwich.

To make this possible, on a technical level, CrabSandwich is composed of two crates: the CrabSandwich runtime, which uses LibAFL to replace and extend LibFuzzer's features, and the component to link this runtime to the fuzz harness and exports the symbols from `libfuzzer-sys`.

3.1 CrabSandwich Runtime

The runtime component of CrabSandwich is a bytes input-based LibAFL fuzzer runtime. To implement the required compatibility features and to leverage the existing components of LibAFL, we implement a range of novel components in CrabSandwich.

General LibFuzzer *Compatiblity.* To remain compatible with cargo-fuzz and fuzz infrastructure for existing harnesses, CrabSandwich utilizes LibAFL *Observers* specifically designed to replace LibFuzzer and LLVM's SanitizerCoverage. We restrict them to features present in the original LibFuzzer-compatible compilation, retrofitting most of LibAFL's advanced features to the preselected instrumentation options. CrabSandwich needs to accept the command-line flags in the same format and style as LibFuzzer. This dictates what functionality must be present. For CrabSandwich, we implemented supports for all flags that are immediately used by cargo-fuzz, as well as other common flags for convenience in evaluation (namely, `-ignore_crashes` and related flags to ensure that the fuzzer would continue after the first discovered crash during evaluation).

Like LibFuzzer, CrabSandwich assumes that the harness will return to a clean state after each target execution and uses in-process fuzzing. In this form of fuzzing, a harness function is invoked in a loop rather than forking and resetting the state on every execution, reducing the overhead of the runtime's invocation of the harness. To accomplish this, CrabSandwich uses the `InProcessExecutor` wrapped in a `TimeoutExecutor` to invoke the locally exported `LLVMFuzzerTestOneInput` harness function. In the case of a timeout, out-of-memory, or crash (by explicit panic or otherwise), LibAFL's crash handler will emit the testcase in the format specified by LibFuzzer (prefixed by the crash cause and followed by the hash of the input) before either completely halting (if `-ignore_crashes` or its counterparts are unset or `-fork=...` is unset) or committing the current state of the fuzzer to a shared memory region before restoring from it and continuing.

`LLVMFuzzerCustomMutator` *Support.* One critical feature of LibFuzzer is the ability to use custom mutators, implemented in the fuzzer harness, to specialize mutations when performing structural fuzzing or fuzzing a target for which the input is expected in a specific format. This feature is also supported by `libfuzzer-sys`, so CrabSandwich will conditionally invoke `LLVMFuzzerCustomMutator` and `LLVMFuzzerCustomCrossover` instead of the built-in mutators, exposing the built-in mutators via `LLVMFuzzerMutate`. In addition, CrabSandwich ensures that inputs are only mutated according to the custom mutation strategies, applied by conditionally configuring which mutations are available depending on which custom mutators are detected, something not present in LibFuzzer.

Harness Input Rejections. LibFuzzer harnesses are permitted to explicitly emit an exit code which specifies whether to retain or reject the most recently executed input in order to only retain inputs which guide towards the desired region [29]. To accomplish this, a rather straightforward observer component was implemented to observe the return value of the harness, which is used in part to determine whether an input will be considered for addition to the corpus.

3.2 Advanced Features

Apart from general LibFuzzer compatibility, we improve fuzzer efficiency by using strong mutation scheduling. As a long-time leader in FuzzBench [36], AFL++ has consistently proven to choose very strong strategies for corpus scheduling or the selection of which inputs to mutate next. Hence, rather than recreating the "entropic" scheduling strategy as LibFuzzer, CrabSandwich uses LibAFL components which implement AFL++-style power scheduling and dynamic minimization of which inputs are available to select. Additionally, we have implemented novel functions to optimize LibFuzzer for Rust fuzzing:

Conditional Grimoire. CrabSandwich is not restricted to just LibFuzzer mutations. A recent paper, Grimoire [10], implements mutations which preserve and mine structure from inputs. Support for Grimoire-style mutations has been present in LibAFL for some time. Since Grimoire mutations are primarily useful for targets which expect structured string inputs, CrabSandwich conditionally enables these mutations depending on the fraction of valid UTF-8 inputs present in the corpus at initialization, and also permits the user to override this setting via a flag.

Comparison Interceptors and String-oriented Mutations. LibFuzzer intercepts multiple common comparison functions in

order to detect longer comparison data than what is available via comparison log [31]. While not implemented in the initial version of CrabSandwich, we intend to implement and evaluate this addition for the full implementation. In addition to using comparison interceptors to collect longer binary and string data, LibFuzzer additionally implements mutators which target string-based inputs [32]. Historical FuzzBench experiments have indicated that these are quite powerful for targets which accept primarily string-based inputs[1]. As such, CrabSandwich in future iterations will similarly implement string-oriented mutations, which will be evaluated in secondary experiments to determine their efficacy. Where the Grimoire mutations preserve structure [10], these mutations would preserve datatype information (like a mutation applied to a number encoded in a string).

Comparison Log. LibFuzzer is able to use feedback from SanitizerCoverage's comparison tracing to guide mutations. CrabSandwich implements a similar strategy, using the comparison log algorithm from AFL++ to guide mutations.

3.3 `libfuzzer-sys` Replacement Design

Unlike the runtime crate, the `libfuzzer-sys` replacement crate serves primarily as the front-facing interface to the user while guaranteeing that the harness is linked to the runtime crate. Its implementation is comparatively straightforward, as this crate itself merely provides a thin interface to the actual runtime. As previously stated, to make CrabSandwich easily accessible to developers, developers should have as few migration requirements as possible. To support this, CrabSandwich's `libfuzzer-sys` replacement crate itself depends on `libfuzzer-sys`, re-exporting all of its symbols and crate features. In this way, users of CrabSandwich can simply swap out the dependency and require no additional changes to the harness.

Building and Linking CrabSandwich's Runtime. Since cargo-fuzz instruments both the harness and the dependencies of the harness, CrabSandwich's runtime must be built and linked separately. To do so, the build script for the `libfuzzer-sys` replacement crate builds the runtime while discarding the sanitization flags exported by cargo-fuzz, and then links the unsanitized runtime.

Harness Input Rejection. LibFuzzer harnesses are permitted to explicitly emit an exit code which specifies whether to retain or reject the most recently executed input in order to only retain inputs which guide towards the desired region [29]. To accomplish this, a rather straightforward observer component was implemented to observe the return value of the harness, which is used in part to determine whether an input will be considered for addition to the corpus.

Publishing CrabSandwich. To make CrabSandwich widely available and easy to use, we will publish CrabSandwich to the `crates.io` package repository and GitHub. In this way, anyone who wishes to try, inspect, or modify CrabSandwich may easily do so.

4 EVALUATION

In this section, we describe the methodology that we plan to follow for the full-scale experiments and we provide a small preliminary experiment involving just three targets.

4.1 Methodology

To evaluate the performance of CrabSandwich versus the baseline, the vanilla cargo-fuzz using LibFuzzer, we plan to conduct extensive experiments targeting many more Rust targets with different design choices – such as the usage of unsafe code or not, or the presence of custom mutators – and different input formats.

The fuzzers will be tested in terms of uncovered bugs and code coverage. The bugs will be manually deduplicated from the set of crashing testcases produced by the fuzzers and the time to discover the first crashing testcase for each bug will be reported to measure the bug-finding ability of the fuzzers. Each experiment will last 24 hours, a commonly chosen time for fuzzing experiments [25]. To cope with the effect of randomness in fuzzing, we plan to repeat each measurement 10 times computing the mean values to represent each result. The Mann-Whitney U test will be used to verify the statistical significance of the results by comparing differences between the two independent groups of values representing the two evaluated fuzzers.

For the preliminary evaluation of our approach, we focus on just 3 target programs restricting the campaign time to just 12 hours and three trials. The code coverage is reported only at the end of the campaign, while we plan to extend the measure to coverage over time in the final evaluation to understand if a fuzzer is faster than another to reach the same coverage.

4.1.1 Benchmarks. For the preliminary evaluation, we choose 3 common Rust crates with an already available cargo-fuzz harness:

- `image`, version 0.24.6
- `pdf`, version 0.8.1, "cache" feature enabled
- `regex`, version 1.8.1

The `pdf` crate was slightly modified to make it compile with the current version and with an additional function call to explore mode code.

As there is no benchmark suite for fuzzers evaluation with known vulnerabilities in Rust such as Fuzzbench [36] or Magma [22] for C/C++ fuzzing, we will build a benchmark suite from scratch selecting old and vulnerable versions of crates as well as improving existing harnesses for current versions of crates. For the preliminary evaluation, we selected the latest version of such projects to show that CrabSandwich is able to discover novel bugs.

4.1.2 Setup. The preliminary experiments were conducted on a Ubuntu 22.04.2 x86_64 machine with a 12 cores 11th Gen Intel(R) Core(TM) i5-11400 2.60GHz CPU and 32 GB of RAM. The bugs found during the campaigns were manually deduplicated as the number of crashing testcases is not a valid metric to assess the performance of a fuzzer [25]. The coverage was computed using the `cargo fuzz coverage` built-in utility of cargo-fuzz that uses the `-Cinstrument-coverage` flags of rustc to compute line coverage.

[1] Note the high performance of LibFuzzer on inputs which expect string-based inputs versus those which expect binary data inputs: https://www.fuzzbench.com/reports/experimental/2023-04-27-main/index.html

Table 1: Mean corpus size, mean and max unique bugs coverage and mean line coverage after 12h with 3 trials for CrabSandwich and cargo-fuzz over the 3 preliminary benchmarks.

Benchmark	Mean Corpus size		Mean bugs coverage		Max bugs coverage		Mean line coverage	
	CrabSandwich	cargo-fuzz	CrabSandwich	cargo-fuzz	CrabSandwich	cargo-fuzz	CrabSandwich	cargo-fuzz
image	1141.6	1591.3	0	0	0	0	8.25%	8.23%
pdf	630.6	687	5	3	7	5	17.86%	17.42%
regex	62.6	75.6	0	0	0	0	30.06%	34.09%

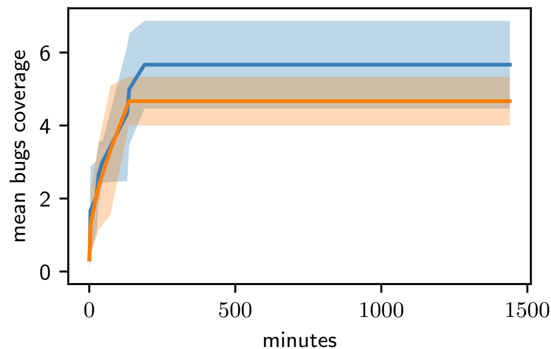

Figure 2: Mean unique bugs coverage and standard error over time until the 12h limit with 3 trials for CrabSandwich and cargo-fuzz on the pdf target benchmark.

4.2 Preliminary Experiments

The preliminary experiments results are summarized in Tab. 1. Only in one benchmark, we could find crashes corresponding to a total of 8 novel bugs that we are in the process of reporting to the project authors. Our variant of cargo-fuzz outperforms the baseline in both the mean and max number of bugs, finding 2 more bugs than the baseline in total. CrabSandwich is also faster in discovering the bugs over time, as plotted in Fig. 2, in which the curve of the mean bugs coverage found by CrabSandwich is always above the baseline.

Among the 8 bugs that we discovered in the `pdf` crate, 5 bugs are caused by index-out-of-range, 2 are panics, and 1 is caused by assertion failure. During the 3 fuzzing campaigns, CrabSandwich discovered 4 of the index-out-of-range bugs, and all of the panics and assertion failures, but missed one index-out-of-range bug.

In terms of code coverage, CrabSandwich is the best performer in 2 out of 3 benchmarks. Being `image` a huge codebase, the small percentage points of difference matters, while the improvement in coverage in `pdf` can explain the additional bugs found. cargo-fuzz is still the best in `regex`, a hard-to-fuzz program as the initial seed is empty and a generic fuzzer has difficulties in generating valid or almost valid regular expressions. LibFuzzer, however, can take advantage of the string-oriented mutations [32], which still are a work-in-progress for CrabSandwich at the time of this preliminary evaluation, but will be available in the final version.

An additional metric that we report is the corpus size, not to assess the performance of the fuzzers in terms of coverage as the number of saved testcases highly depends on the type of feedback used and cannot be used to compare effectiveness, but as a metric useful to understand if a fuzzer is saving too many redundant testcases for the uncovered coverage causing wastage to the internal state of the fuzzing algorithm. In general, CrabSandwich seems to achieve more coverage requiring a smaller corpus, except for `regex` in which the greater corpus size of cargo-fuzz seems to correlate with the additional coverage uncovered.

4.3 Discussion

For our initial evaluation, CrabSandwich had marginal improvements over base cargo-fuzz in two of the targets and worse coverage in one. Despite this, CrabSandwich consistently maintained a smaller corpus size and, for the target with bugs present, found two more bugs than base cargo-fuzz both by mean and max. Simultaneously, it would be unreasonable to state that this evaluation is near complete or sufficient. As discussed in 4.1, the full evaluation will include several additional targets.

In addition, these results support the motivation for implementing comparison interceptors and string-oriented mutations. Observing the behavior of the `regex` fuzzer, LibFuzzer records several strings while running the fuzzer. After some investigations, it becomes clear that it records the names of Unicode character classes. These names are used by the regex crate for matching based on a Unicode character class [46]. As of now, neither CrabSandwich nor LibAFL is able to replicate this behavior as it does not record strings or memory regions passed to comparison functions. As a result, CrabSandwich is unable to cover significant portions of the `regex` crate.

A secondary objective of this work is to develop and improve existing harnesses of Rust libraries, as needed for evaluating CrabSandwich. In the course of our initial evaluation, we discovered *eight* new bugs in the `pdf` crate. A fuzz harness for this target already exists, and yet with minor modifications, we were able to discover several new bugs. In this way, we expect that, in the process of evaluating CrabSandwich, we will develop other additional improvements to harnesses already present in the community, responsibly disclosing to project owners and reporting on our findings in our final submission. Moreover, by standardizing, developing, and improving existing fuzz harnesses for Rust targets, we will inherently create a benchmark suite for Rust-oriented fuzzers.

5 RELATED WORK

In this section, we discuss some related works that focus on Rust vulnerability discovery and advancing the capabilities of C/C++ fuzzers that can be potentially used in combination with our approach.

5.1 Vulnerabilities in Rust

Vulnerabilities in Rust codebases can be usually linked to the usage of `unsafe` code [8]. Developers can use the "unsafe" keyword to go beyond the language's type system and bypass that memory safety guarantees of the language with the Rust compiler that assumes that the provided unsafe code is correct and bug-free. This means that a single bug in unsafe code, whether it originates from the developer's own code or from a library, can destroy the safety guarantees of the entire program and thus the usage of unsafe code patterns must be strictly limited to the necessary usage. In addition, there is always the possibility of having bugs in the Rust type system itself [44] breaking the safety guarantees for the programs compiled with a buggy compiler.

Recent works in vulnerability discovery tried to focus on Rust programs using both static and dynamic approaches. Rudra [8] analyzed statically every package in the online Rust ecosystem reporting more than 200 bugs linked to the usage of unsafe code. On the other hand of the testing spectrum, Rulf [24] address the lack of fuzzing harnesses for libraries with automatic harness generation for Rust crates.

5.2 Modern Fuzzing Advancements

5.2.1 Code Coverage Roadblocks. Code coverage roadblocks are a challenge to code exploration in fuzzing as standard mutations struggle to satisfy comparison sequences within the input. Magic values and checksum fields in input formats are two common types of roadblocks. Magic values involve multi-byte comparisons, while traditional structure-blind mutators treat the input as a byte stream, making it highly improbable to match all the involved bytes. Potential solutions include using specialized feedback for partial progress in comparisons [2, 27] employing concolic execution for white-box fuzzing [12, 50], or techniques that extract comparison operands to replace input segments [4, 40]. Checksums, often used for validation in binary formats, pose even greater difficulty. Existing solutions involve format-specific mutators or code transformations that temporarily override checksum checks [4, 38].

5.2.2 Input Generation. General-purpose fuzzers often generate a high proportion of invalid inputs, which tend to exercise code only in the early parsing stages of a program, failing to explore deeper regions. To effectively explore these regions, a fuzzer must prioritize producing valid inputs. Some approaches, like [3, 39], utilize hand-written input format specifications to guide the mutator. More recent works have attempted to automatically learn approximate specifications [9, 10, 16].

5.2.3 Bug Oracles. While most fuzzers aim to expose bugs that may indicate potential vulnerabilities, some bugs do not immediately trigger observable crashes. To catch such bugs, fuzzing users instrument the program under test with additional checks to detect the silent bugs with enhanced oracles. Source-based fuzzers, for instance, offer the option to instrument programs with sanitizers like ASan [42]. However, current sanitizers may not capture certain pure-logic bugs, and automatically uncovering such bugs through fuzzing, without manual assertions, remains an active area of research.

6 CONCLUSION

As part of the extended study, we hope to further prove that Rust is the perfect fit to fuzz Rust programs. CrabSandwich is still in an early stage of development and will improve over time, driven by advancements in the underlying LibAFL. In contrast to the original cargo-fuzz runtime, LibFuzzer, the preliminary results look very promising, and we are confident that future evaluations will continue this positive trend. The results of the evaluation support the motivation for implementing comparison interceptors and string-oriented mutations. These features allow CrabSandwich to find bugs that cargo-fuzz cannot, and they are likely to be even more effective when used in conjunction with other fuzzing techniques.

Overall, the preliminary evaluation of CrabSandwich is promising. We have made significant progress in developing a more effective fuzzer for Rust, and we have identified several areas for future improvement. With further development, CrabSandwich has the potential to be a valuable tool for finding an increasing amount of bugs in Rust software.

All tooling we develop will be open-sourced upon publication of this paper.

ACKNOWLEDGEMENTS

We want to thank the community around the AFL++ organization, especially the LibAFL contributors. We would like to thank also the anonymous reviewers and Slasti Mormanti for the valuable suggestions.

REFERENCES

[1] (n. d.). Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz. [Online; accessed 10 May. 2023].
[2] 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/. [Online; accessed 10 May. 2023].
[3] Cornelius Aschermann, Tommaso Frassetto, T. Holz, Patrick Jauernig, A. Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *NDSS*.
[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/
[5] Rust Fuzzing Authority. (n. d.). afl.rs: Fuzzing Rust code with American Fuzzy Lop. https://github.com/rust-fuzz/afl.rs. [Online; accessed 10 May. 2023].
[6] Rust Fuzzing Authority. (n. d.). cargo-fuzz: Command line helpers for fuzzing. https://github.com/rust-fuzz/cargo-fuzz. [Online; accessed 10 May. 2023].
[7] Rust Fuzzing Authority. (n. d.). honggfuzz-rs: Fuzz your Rust code with Google-developed Honggfuzz ! https://github.com/rust-fuzz/honggfuzz-rs. [Online; accessed 10 May. 2023].
[8] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: finding memory safety bugs in Rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.
[9] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. 2023. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *32st USENIX Security Symposium (USENIX Security 23)*. USENIX Association.
[10] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko
[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428
[12] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. https://doi.org/10.1109/ICSE43902.2021.00071
[13] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*. 18–35. https://doi.org/10.1109/SP46214.2022.9833796

[14] M. Eddington. (n. d.). Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed 10 May. 2023].

[15] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2829–2846. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[16] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3395363.3397372

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[18] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)* (Los Angeles, U.S.A.) *(CCS '22)*. ACM.

[19] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. 2023. Dissecting American Fuzzy Lop: A FuzzBench Evaluation. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 52 (mar 2023), 26 pages. https://doi.org/10.1145/3580596

[20] Patrice Godefroid. 2007. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. 1–1.

[21] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/fuzzilli-fuzzing-for-javascript-jit-compiler-vulnerabilities/

[22] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428334

[23] Adrian Herrera, Mathias Payer, and Antony L. Hosking. 2023. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Trans. Softw. Eng. Methodol.* (mar 2023). https://doi.org/10.1145/3587156 Just Accepted.

[24] Jia Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust Library Fuzzing via API Dependency Graph Traversal. *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), 581–592.

[25] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. https://doi.org/10.1145/3243734.3243804

[26] LLVM. (n. d.). SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html. [Online; accessed 10 May. 2023].

[27] LLVM Project. (n. d.). libFuzzer - Value Profile. https://llvm.org/docs/LibFuzzer.html#value-profile. [Online; accessed 10 May. 2023].

[28] LLVM Project. 2018. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html. [Online; accessed 10 May. 2023].

[29] LLVM Project. 2023. libFuzzer – Rejecting unwanted inputs. https://llvm.org/docs/LibFuzzer.html#rejecting-unwanted-inputs. [Online; accessed 10 May. 2023].

[30] LLVM Project. 2023. libFuzzer – Status. https://llvm.org/docs/LibFuzzer.html#status. [Online; accessed 10 May. 2023].

[31] LLVM Project and Rust Fuzzing Authority. 2022. rust-fuzz/libfuzzer – FuzzerInterceptors.cpp. https://github.com/rust-fuzz/libfuzzer/blob/1221c356e993b9f82d1ccd152f1c7636468758d2/libfuzzer/FuzzerInterceptors.cpp. [Online; accessed 10 May. 2023].

[32] LLVM Project and Rust Fuzzing Authority. 2022. rust-fuzz/libfuzzer – FuzzerMutate.cpp. https://github.com/rust-fuzz/libfuzzer/blob/main/libfuzzer/FuzzerMutate.cpp. [Online; accessed 10 May. 2023].

[33] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[34] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. 2022. FitM: Binary-Only Coverage-Guided Fuzzing for Stateful Network Protocols. In *Workshop on Binary Analysis Research (BAR), 2022*.

[35] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. 2022. Fuzzing with data dependency information. In *EuroS&P 2022, 7th IEEE European Symposium on Security and Privacy, 6-10 June 2022, Genoa, Italy*, IEEE (Ed.). Genoa.

[36] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA.

[37] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[38] H. Peng, Y. Shoshitaishvili, and M. Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. 697–710. https://doi.org/10.1109/SP.2018.00056

[39] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[41] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Vancouver, B.C. https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, 28.

[43] Dongdong She, Abhishek Shah, and Suman Sekhar Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. *2022 IEEE Symposium on Security and Privacy (SP)* (2022), 2194–2211.

[44] Frank Steffahn. (n. d.). rust-lang/rust: Unsoundness in type checking of trait impls. Differences in implied lifetime bounds are not considered. https://github.com/rust-lang/rust/issues/80176. [Online; accessed 10 May. 2023].

[45] Robert Swiecki. (n. d.). Honggfuzz. https://github.com/google/honggfuzz. [Online; accessed 10 May. 2023].

[46] The Rust Project Developers. 2023. regex - Rust – Matching One Character. https://docs.rs/regex/latest/regex/#matching-one-character. [Online; accessed 10 May. 2023].

[47] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[48] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/reinforcement-learning-based-hierarchical-seed-scheduling-for-greybox-fuzzing/

[49] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.

[50] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 745–761.

[51] Michał Zalewski. 2016. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. [Online; accessed 10 May. 2023].