# Don't Panic! Finding Bugs Hidden Behind Rust Runtime Safety Checks

Zeyang Zhuang
The Chinese University of Hong Kong
Hong Kong SAR, China
zyzhuang22@cse.cuhk.edu.hk

Zilun Wang
The Chinese University of Hong Kong
Hong Kong SAR, China
zlwang22@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong SAR, China
lyu@cse.cuhk.edu.hk

## Abstract

Rust has been extensively used in software and system development due to its guarantees for memory and concurrency safety. Fuzzing is a popular bug detection technique for examining the correctness and robustness of programs. However, we identify that current state-of-the-art Rust fuzzers are significantly impeded by the ubiquitous presence of Rust runtime safety checks, resulting in poor effectiveness and efficiency. These checks, which are inserted either implicitly by the compiler or explicitly by the compiler or developers, could cause a high number of panic crashes and early program termination in fuzzing. Consequently, current fuzzers are unable to effectively explore deep code behind the runtime safety checks, leaving potential vulnerabilities undetected.

To address these limitations, we propose PanicKiller, a new Rust fuzzing technique to detect bugs hidden in deep and unsafe code. It performs a cross-IR analysis to precisely identify runtime safety checks and unsafe code in Rust programs, and employs a novel dynamic taint analysis to track the critical input bytes associated with the conditions enforced by these checks. PanicKiller further performs novel input prioritization and mutation strategies to achieve effective and efficient fuzzing. Our evaluation shows that PanicKiller significantly outperformed current state-of-the-art Rust fuzzers by achieving average improvements of 22.0× in bug exposure speed, 1.68× in code coverage, and 18.2× in false-positive crash reduction, and up to 129.0×, 2.10×, and 64.8× improvements, respectively. PanicKiller further helped detect 14 and 53 previously unknown vulnerabilities in the benchmark dataset and in the real world, with 11 RustSec IDs assigned.

## CCS Concepts

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

## Keywords

Fuzzing; Software Testing; Rust Security

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.
*CCS '25, Taipei, Taiwan*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765142

**ACM Reference Format:**
Zeyang Zhuang, Zilun Wang, Wei Meng, and Michael R. Lyu. 2025. Don't Panic! Finding Bugs Hidden Behind Rust Runtime Safety Checks. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765142

## 1 Introduction

Rust is a modern programming language designed for developing reliable and efficient software and systems. It strictly enforces memory safety and thread safety to provide safety guarantees. Since memory safety issues in languages such as C and C++ have led to severe consequences and are difficult to prevent, Rust becomes a good system programming language alternative. It has been integrated into prestigious open-source projects, such as the Linux kernel [34] and the Firefox browser [33]. The United States government has also advocated for the adoption of memory safety languages such as Rust to secure the nation's cyberspace [28].

Rust performs stringent safety checks in the safe code [24], namely the compile-time safety checks and runtime safety checks, to ensure memory safety and concurrency safety. Rust compile-time safety checks [29] incorporate the ownership and borrow checks, *etc.*, to eliminate many classes of bugs before program execution without affecting the runtime behaviors. Rust runtime safety checks [41] serve as a complement to detect and mitigate undefined behaviors and errors during program execution. Nevertheless, to provide more programming flexibility, Rust allows developers to write unsafe code for direct memory manipulation and performance-critical tasks. Existing studies [54, 70] have demonstrated that unsafe code is the primary cause of memory safety issues in Rust programs.

Fuzzing is a widely adopted technique to detect software bugs. Current state-of-the-art Rust fuzzers [12, 13, 30] integrate the existing LLVM-based fuzzers [15, 19, 51] for bug detection. However, we identify that these Rust fuzzers are unaware of Rust runtime safety checks that enforce safety guarantees during execution, and would generate numerous inputs that fail these checks and cause the program to panic, thereby resulting in poor effectiveness and efficiency. To substantiate this observation, we conducted a study to evaluate current state-of-the-art Rust fuzzers—afl.rs [13], cargo-fuzz [30], and honggfuzz.rs [12]. We built a benchmark dataset consisting of 16 Rust packages with known bugs, which were selected from the evaluation results in recent Rust library fuzzing

research [56, 71, 72]. We reveal that for every detected true-positive bug, afl.rs, cargo-fuzz, and honggfuzz.rs triggered on average 180, 200, and 210 false-positive crashes (*e.g.*, panics), which were caused by the failures of runtime safety checks. These false positives prolong the time to expose true bugs and limit code coverage. We further develop a static analyzer to identify and assess the impacts of these runtime safety checks in the programs. We observe that Rust runtime safety checks are extensively inserted along most execution paths, and programs with more runtime safety checks impose a greater hindrance on fuzzing performance.

Through our study, we identify two limitations of the existing Rust fuzzers. First, their effectiveness is limited by the high false positives they report. Many crashes detected by these fuzzers are panics when runtime safety checks fail, and are intended behaviors. This reduces the effectiveness of the fuzzers and requires substantial (manual) effort to filter out irrelevant results. Second, early termination leads to false negatives. These fuzzers generate inputs that frequently fail runtime safety checks. This causes programs to early panic, preventing exploration of deep code where potential bugs remain undiscovered.

These findings motivate us to develop a new fuzzing technique to detect the deep bugs hidden behind the Rust runtime safety checks and bugs stemming from unsafe code. However, there are two primary challenges in achieving our goals. First, it is non-trivial to precisely identify runtime safety checks and unsafe code for LLVM-based fuzzers, as such semantic information is refactored or optimized out in LLVM IR. Second, it is challenging to generate inputs to pass runtime safety checks. Rust runtime safety checks enforce various conditions, including those related to arithmetic operations, and memory access, *etc.* To pass these checks, certain bytes in the input need to adhere to the associated conditions, such as bounded integer ranges, non-zero values, and other valid range requirements, *etc.* However, current Rust fuzzers cannot address these challenges. Existing constraint solving techniques can hardly tackle this program either. For example, concolic execution [1] suffers from the path explosion issue when handling many runtime safety checks, while existing intelligent branch solving approaches [49, 62] are limited to C and C++ programs and do not apply to Rust.

In this paper, we propose PanicKiller, a new Rust fuzzing technique to detect deep bugs hidden behind runtime safety checks. Our main idea is to generate inputs to pass the checks, thereby avoiding the panics and enabling deeper code exploration. To this end, PanicKiller accurately identifies runtime safety checks and the conditions they enforce, and subsequently leverages the information to pass the checks and explore deep and unsafe code. Specifically, PanicKiller performs a new *cross-IR analysis* to precisely identify the semantic information—such as runtime safety checks, unsafe code, and Rust types—that can only be determined in Rust-specific IRs. It then propagates such information to the corresponding LLVM IR instructions for fuzzing. Since different execution paths may contain multiple and various categories of runtime safety checks, we aim to generate inputs tailored to them. We propose a novel *dynamic taint analysis* technique to identify the critical input bytes that are associated with the conditions enforced by the runtime safety checks. We also infer the relevant semantic information of the runtime safety checks, such as bounded integer ranges, types, *etc.* PanicKiller incorporates new mutation strategies to improve the likelihood of passing runtime safety checks, for example, by applying linear mutation on critical bytes, and by constraining the values of the critical bytes based on the types to mitigate overflow issues. Moreover, PanicKiller schedules fuzzing energy in an efficient manner. For execution paths with runtime safety checks, PanicKiller performs novel *input prioritization* to prefer those with fewer checks. It also prioritizes in testing unsafe code to discover potential bugs.

We extensively evaluated the performance of PanicKiller using our benchmark dataset. PanicKiller detected all the bugs in the dataset, and additionally detected 14 new vulnerabilities that the other fuzzers cannot detect. The 14 new vulnerabilities include five dynamic stack buffer overflows, seven segmentation faults on unknown addresses, and two bugs of stack use after scope. Our evaluation results demonstrate that PanicKiller significantly outperformed all three state-of-the-art Rust fuzzers—afl.rs, cargo-fuzz, and honggfuzz.rs—across multiple aspects: it achieved average improvements of 22.0× in bug exposure speed, 1.68× in code coverage, and 18.2× in false-positive crash reduction, with maximum improvements of 129.5×, 2.1×, and 64.8×, respectively. We further demonstrate the effectiveness of each new technique we propose with a component-wise evaluation. They enable PanicKiller to pass runtime safety checks, while allocating more energy towards detecting bugs in deep and unsafe code. We further applied PanicKiller to detect unknown bugs in the real world for its superb fuzzing performance. It identified 53 new vulnerabilities in 22 Rust packages. Specifically, it detected 24 stack and heap overflows, one bug of stack use after scope, eight segmentation faults on unknown addresses, three bugs of invalid allocation alignment, and several program aborts, *etc.*, which can pose severe security risks. We responsibly disclosed all our findings to the relevant developers. At the time of writing, 38 vulnerabilities have been confirmed by the developers, five have been timely fixed, and 11 new RustSec IDs have been assigned.

In summary, we make the following contributions:

- We present an in-depth study to demonstrate how Rust runtime safety checks impede the fuzzing effectiveness and efficiency, and reveal that current state-of-the-art Rust fuzzers perform poorly.
- We develop a novel Rust fuzzer, PanicKiller, that specializes in detecting deep bugs behind the runtime safety checks.
- PanicKiller achieved superb performance compared to current state-of-the-art Rust fuzzers in the evaluation.
- With PanicKiller, we have detected 14 and 53 previously unknown vulnerabilities in a benchmark dataset and in the real world, respectively. We have been assigned 11 RustSec IDs.

## 2 Background

In this section, we present the fundamental relevant knowledge.

### 2.1 Safe and Unsafe Rust

Rust clearly distinguishes between safe and unsafe code. Safe Rust enforces stringent safety checks to achieve safety guarantees, which we discuss further in §2.2. In contrast, unsafe Rust permits operations that violate the Rust safety checks, transferring the responsibility for memory safety to developers. Unsafe Rust allows five

specific abilities [17], namely dereferencing raw pointers, calling unsafe functions, altering mutable static variables, implementing unsafe traits, and accessing fields of unions. Existing studies [54, 70] have shown that unsafe code is the most common cause of memory safety issues in Rust programs.

## 2.2 Rust Safety Checks

Rust ensures memory safety and thread safety while preventing common programming errors through rigorous safety checks. There are mainly two safety check mechanisms in Rust, namely compile-time checks [29] statically enforced by the compiler, and runtime checks [41] dynamically performed during program execution.

**Compile-time safety checks.** Rust compile-time safety checks incorporate features and checks [9] like ownership, borrowing, and lifetime, *etc.* to prevent memory errors and data races, while enforcing strict type checking [44]. When a program fails these compile-time checks, the compiler will terminate compilation and emit error messages for the developers to address the safety issues. This mechanism effectively enables developers to eliminate and fix many classes of errors and bugs at compile time, and would not affect the program's runtime behaviors.

**Runtime safety checks.** Rust runtime safety checks serve as a complement to compile-time safety checks by dynamically mitigating undefined behaviors and errors that are not detected at compile time. This mechanism is achieved through Rust's built-in `panic!` macro [23]. When the runtime safety checks detect violations, the panic signals that the program is in an invalid state. The panic handling code will perform stack unwinding and cleanup, terminate the program, and prevent the errors from propagating. There are two primary methods to trigger a panic [41]. First, the Rust compiler designs several panic-related language items [35] to enforce runtime safety checks, such as checks for array out-of-bound access. The Rust standard libraries also implement runtime safety checks based on the language items, such as `slice` and `vec` checks in the `alloc` library, and `unwrap` checks in the `core` library, *etc.* For example, at lines 3-9 in Listing 1, there are several runtime safety checks, *e.g.*, bounds check, overflow check, and `unwrap` check, inserted by the Rust compiler and standard libraries. Second, developers can design runtime safety checks according to their intentions by explicitly calling macros such as `panic!` or `assert!` [22] within their code. At lines 10-11, the developer inserts a customized assert check for safety considerations.

## 2.3 Rust Fuzzing

Fuzzing is a widely used bug finding approach. Current state-of-the-art Rust fuzzers [12, 13, 30] integrate LLVM-based fuzzers [63] for bug detection. However, we identify that these fuzzers [12, 13, 30] could generate numerous test inputs that fail runtime safety checks, which significantly impedes the effectiveness and efficiency of fuzzing. The reasons are twofold. First, many detected crashes are essentially false positive memory safety problems. These crashes occur due to violations of the safety properties enforced by runtime safety checks, thereby reducing fuzzing effectiveness. The Rust developers [32] and prior Rust library fuzzing research [56, 71, 72] regard the panic crashes triggered by runtime safety checks as intended behaviors, and false positives in fuzzing. Like stack canaries [37], these checks are placed to prevent further exploitation of runtime memory safety issues. As a result, such panics are often ignored by developers and rarely fixed. Second, the Rust programs under testing are frequently terminated prematurely due to the panics, preventing exploration of deep code and leaving hidden bugs undiscovered. This significantly degrades the fuzzing efficiency, as substantial resources are spent on trying to pass the runtime safety checks. We further use our real-world study in §3 to demonstrate the limitations of existing fuzzers.

```rust
1  fn main() {
2      let val: usize = std::io::stdin().read_line(&mut i).
        unwrap(); // Runtime input
3      // Bounds check
4      let array: Vec<u8> = vec![1, 2, 3, 4, 5];
5      let _ = array[val];
6      // Overflow check
7      let _ = val + std::usize::MAX - 6;
8      // Unwrap check
9      let _ = array.get(val).unwrap();
10     // Customized check
11     assert!(val < 10, "val should be less than 10");
12     // More runtime safety checks...
13     // Deep code and bugs...
14 }
```

**Listing 1: An example showing that in order to reach deep code, the input needs to pass multiple runtime safety checks.**

# 3 A Motivating Study

We first discuss our study of existing Rust fuzzers and their limitations in §3.1. We then use a static analyzer we designed to demonstrate that the prevalence of runtime safety checks can impede fuzzing performance in §3.2

## 3.1 Existing Rust Fuzzers

We identify that none of the state-of-the-art Rust fuzzers is aware of the significance of passing runtime safety checks. To evaluate all the current Rust fuzzers, we construct a real-world benchmark dataset and present the evaluation results with deduplication [4] in Table 1. We select Rust packages with known bugs based on the evaluation results from recent Rust library fuzzing research [56, 71, 72] to construct this benchmark dataset. Our benchmarking results show that for every true-positive bug detected, afl.rs, cargo-fuzz, and honggfuzz.rs triggered 180, 200, and 210 false-positive crashes on average, with maximums of 1,550, 1,650, and 1,950 crashes (*e.g.*, in the regex package), respectively. We also observe that failures of runtime safety checks would prolong the time to expose true bugs and restrict code coverage. We discuss more details of our experiments in §7.1. Our results are consistent with prior work [56, 71, 72]. They similarly triggered a large amount of false-positive crashes during experiments, but they did not address the issues.

We identify two key limitations of current Rust fuzzers. *First, their effectiveness is limited by many false positives they report.* The LLVM-based fuzzers, which are primarily designed for C and C++ programs, interpret crashes as memory safety bugs. In contrast, the panic crashes stemming from Rust runtime safety checks are

**Table 1: Evaluation results on the benchmark dataset. All checks and average checks refer to the total number and the average number of runtime safety checks among all paths towards the bugs.**

| Package | afl.rs | cargo-fuzz | honggfuzz.rs | All | Average |
|---|---|---|---|---|---|
| | Crashes/Bugs | Crashes/Bugs | Crashes/Bugs | Checks | Checks |
| autocfg-1.0.1 | 8/1 | 10/1 | 8/1 | 34 | 2 |
| bumpalo-3.9.1 | 24/2 | 22/2 | 25/2 | 96 | 8 |
| byteorder-1.4.3 | 286/6 | 367/6 | 280/6 | 120 | 12 |
| bytes-1.1.0 | 225/4 | 314/4 | 264/4 | 224 | 14 |
| clap-2.33.3 | 1078/2 | 1254/2 | 1338/2 | 688 | 86 |
| fixedbitset-0.4.1 | 56/2 | 70/2 | 70/2 | 90 | 10 |
| flate2-1.0.19 | 11/1 | 12/1 | 13/1 | 32 | 2 |
| mio-0.8.0 | 6/1 | 7/1 | 7/1 | 48 | 4 |
| proc-macro2-1.0.24 | 103/1 | 142/1 | 107/1 | 240 | 30 |
| regex-1.5.4 | 4750/2 | 4986/2 | 5830/2 | 552 | 184 |
| csv-1.1.6 | 130/2 | 129/2 | 137/2 | 312 | 26 |
| url-2.2.0 | 422/2 | 440/2 | 479/2 | 360 | 40 |
| ryu-1.0.9 | 31/1 | 40/1 | 39/1 | 126 | 14 |
| time-0.2.24 | 321/5 | 326/5 | 411/5 | 232 | 29 |
| uni-seg-1.8.0 | 54/10 | 65/10 | 56/10 | 84 | 6 |
| uuid-1.4.1 | 60/1 | 65/1 | 69/1 | 144 | 18 |

**Table 2: Classification on the Rust runtime safety checks, including source code examples and the simplified corresponding checks (*i.e.*, the terminator instructions) inserted at the Rust MIR.**

| Runtime safety checks | Code examples | Inserted checks in MIR |
|---|---|---|
| Arithmetic (*e.g.*, Add, Sub, Mul, Div, Rem, Neg, Shr, Shl) overflow | `u8:MAX + 1;` | `"attempt to compute `{$left} {$op} {$right}`, which would overflow" -> [success: bb1, unwind: bb2];` |
| Bounds check (*e.g.*, array, slice, vec checks) | `array[len + 1..]; &s[1..]; //when s="™"` | `"index out of bounds: the length is {$len} but the index is {$index}" -> bb1;` |
| Divide/Remainder by zero | `a / b; //when b = 0` | `"attempt to divide `{$val}` by zero" -> [success: bb1, unwind: bb2];` |
| Unwrap on None/Err value | `x.unwrap(); // when x = None/Err` | `core::panicking::panic(const "called `Option::unwrap()` on a `None` value");` |
| Others: `panic!`, `assert!`, `unreachable!`, *etc.* | `panic!("A panic!"); assert!(1 == 0);` | `begin_panic::<&str>(const "A panic!");` |

intended safeguards. Current Rust fuzzers merely integrate the functionalities of existing LLVM-based fuzzers, without recognizing this issue. Failures of runtime safety checks would result in numerous false positives, making fuzzing less effective and requiring significant effort to filter irrelevant results. *Second, early termination leads to false negatives.* The test inputs generated by current Rust fuzzers frequently fail Rust runtime safety checks, causing programs to early panic before deep code is reached. As a consequence, the bugs that only manifest themselves in the deep code after passing numerous runtime safety checks are difficult to detect. Frequent early termination not only hinders potential bug discovery and code coverage, but also reduces the fuzzing efficiency.

## 3.2 Prevalence of Runtime Safety Checks

To conduct a comprehensive study on the prevalence and impacts of Rust runtime safety checks, we develop a static analyzer to precisely identify the runtime safety checks that are inserted both explicitly and implicitly. We find that programs with more runtime safety checks impose a greater hindrance on fuzzing performance, and the bugs detected are typically preceded by numerous runtime safety checks. In the following, we first introduce the Rust runtime safety checks studied in this paper, and conclude with our findings.

Based on the benchmarking results in §3.1 and a detailed study of Rust language features [6, 35], we classify Rust runtime safety checks into five categories, as shown in Table 2. Specifically, arithmetic overflow checks, bounds checks, divide/remainder by zero checks, and `unwrap` checks are implicitly inserted into programs by the Rust compiler and standard libraries, and are not visible in the source code. In contrast, other customized checks (*e.g.*, `panic`, `assert`, and `unreachable` checks) are explicitly inserted in the source code by the compiler or developers. As Rust evolves continuously, there may exist other categories of Rust runtime safety checks, such as checks related to concurrency, checks for new features [46], *etc.* The implementation of these runtime safety checks also fundamentally relies on the `panic` macro, which we have considered. We leave their analysis as future work. In our benchmark experiments in §3.1, false-positive crashes triggered by these five categories of runtime safety checks account for 100%. Therefore, we have covered the most common runtime safety checks.

Our goal is to compute the number of explicit and implicit Rust runtime safety checks in the programs to quantitatively assess their impact. We discuss the design of our static analyzer in Appendix A. We present the static analysis results on the benchmark dataset in Table 1. The results indicate that Rust packages with more runtime safety checks would potentially cause current Rust fuzzers to trigger more crashes. We also measure that for all paths reaching the bugs in our benchmark dataset, at least 90% of the paths have runtime safety checks. For instance, in the bytes package, there are 16 distinct paths leading to a bug location; among these, 15 paths contain runtime safety checks, with an average of 14 checks, a maximum of 22 checks, and a total of 224 checks. This leads to around 200 false-positive crashes detected by afl.rs. Our results demonstrate that runtime safety checks are extensively inserted both explicitly and implicitly in the programs. Consequently, the inputs generated by Rust fuzzers need to pass as many such checks as possible. Otherwise, they would result in numerous crashes, thereby reducing the fuzzing effectiveness and efficiency.

# 4 Problem Statement

In this section, we discuss our research goals and challenges.

## 4.1 Research Goals and Scope

This paper aims to develop a new effective Rust fuzzing technique for detecting deep bugs behind the Rust runtime safety checks. We consider a code location deep if it is only accessible through passing at least one runtime safety check. Excessive failures of runtime safety checks would significantly limit fuzzing performance. One might attempt to disable or remove Rust runtime safety checks in fuzzing for reaching deeper code locations. However, disabling these checks violates Rust safety properties and may result in false positive crashes caused by unrealistic or invalid data flows.

We seek to examine in detail the unique features of each type of runtime safety checks, including the enforced conditions, *etc.*, and subsequently leverage the information to enhance deep bug detection performance. We define the conditions imposed by the

runtime safety checks as safety conditions. In this work, we focus on passing the runtime safety checks listed in Table 2. We consider other potential runtime safety checks as future work, and plan to enable other researchers to easily extend our design to support new checks. Note that Rust compile-time safety checks introduced in §2.2 do not affect fuzzing performance, because programs that fail compile-time checks cannot be successfully compiled, and consequently cannot be fuzzed.

One may argue that runtime safety checks can potentially expose functionality bugs, as they can prompt developers to improve the robustness of their code, such as sanitizing the runtime inputs. Nevertheless, such issues are not security critical, and many developers choose not to fix. Our benchmarking results in §3.1 also show that the panic crashes caused by runtime safety checks are all false positive memory safety issues. Furthermore, we believe the probability of false negatives due to passing runtime safety checks is essentially negligible for two reasons. First, our bug oracles rely on Address Sanitizer [67], and its memory bug detection capabilities encompass those of Rust runtime safety checks. Second, our fuzzer is designed to avoid failures of runtime safety checks rather than eliminating the checks completely. Therefore, our approach is unlikely to result in false negatives.

## 4.2 Research Challenges

There exist several challenges in achieving our research goals. *First, it is non-trivial for LLVM-based fuzzers to precisely identify runtime safety checks as well as other information like unsafe code in Rust programs.* Such semantic information is available at Rust-specific IRs [14, 25], where many Rust features are still preserved, but is often refactored or optimized out at the LLVM IR level. Specifically, at the LLVM IR, the runtime safety checks are restructured with other instructions and variables, while the information like unsafe code and Rust rich types is eliminated. Besides, there may exist numerous and diverse categories of runtime safety checks across all possible execution paths. It is difficult to accurately identify all the checks and other information by interpreting and analyzing the complex semantics of Rust-specific instructions, and subsequently propagate this information into lower-level LLVM IR instructions that fuzzers can recognize and utilize.

*Second, it is challenging to generate inputs that pass runtime safety checks.* Rust runtime safety checks typically enforce various safety conditions, such as those related to arithmetic operations, memory access, and control flow, *etc.* To pass these checks, certain bytes in the input need to conform to the safety conditions, such as bounded integer ranges, non-zero values, and other valid ranges, *etc.* However, existing Rust fuzzers would easily generate random inputs that cannot meet these conditions. Existing constraint solving methods such as concolic execution and intelligent branch solving can also hardly tackle this problem. Given that a program may contain more than hundreds of runtime safety checks, concolic execution [1] would suffer from critical limitations such as path explosion, and tend to be time-consuming and resource-intensive. Furthermore, current intelligent branch solving approaches (*e.g.*, AFL++ - Cmplog [51], Angora [49]) only handle a predefined set of conditions for C and C++ programs, and are not applicable to Rust. Thus, a novel solution needs to be proposed and developed.

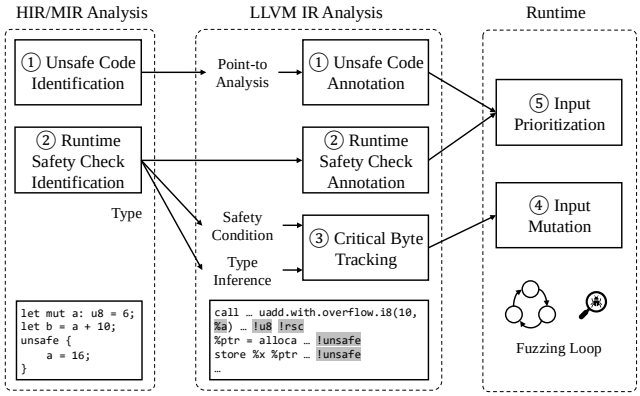


**Figure 1: The architecture of PanicKiller.**

# 5 Design

In this section, we present PanicKiller, a new Rust fuzzing technique for effectively and efficiently detecting deep bugs behind Rust runtime safety checks.

## 5.1 Overview

The architecture of PanicKiller is shown in Figure 1. To address the limitations of existing works and the research challenges, our key insight is to leverage information available in Rust-specific IRs, but not precisely identifiable in LLVM IR, to guide the fuzzer in passing runtime safety checks and exploring deep and unsafe code.

The system mainly consists of three parts. First, it is essential to precisely identify the runtime safety checks and the safety conditions they enforce, as these need to be satisfied during execution. We should also identify information like unsafe code, which may contain undiscovered bugs. Therefore, PanicKiller performs a *cross-IR analysis* (①, ② in Figure 1) to precisely identify the semantic information that can only be determined in Rust-specific IRs, and propagate to LLVM IR for later fuzzing. Second, since different execution paths may contain numerous and various categories of runtime safety checks, we aim to generate high-quality inputs tailored to different checks. To this end, we propose a novel *dynamic taint analysis* (③, ④) to identify the critical input bytes that influence the safety check conditions; we also infer the relevant semantic information of the safety checks, such as bounded integer ranges, types, *etc.* These help guide the fuzzer to generate and mutate inputs more effectively to pass the checks, for example, by constraining the values of the input bytes based on the types. Third, since different paths may contain varying numbers of runtime safety checks, we need to schedule fuzzing energy efficiently. Correspondingly, for execution paths with runtime safety checks, PanicKiller performs *input prioritization* (⑤) to prefer those with fewer checks. PanicKiller also prioritizes in testing unsafe code to discover potential bugs. We discuss these three components in §5.2, §5.3, §5.4, respectively, and finally present PanicKiller's fuzzing loop in §5.5.

## 5.2 Cross-IR Analysis

We aim to provide additional program semantic information to the LLVM IR for the fuzzers to utilize and improve the fuzzing process.

We need to precisely identify the Rust runtime safety checks as well as their associated safety conditions. Besides, it is also necessary to identify the unsafe code in Rust to guide the fuzzers exploring the unsafe code. However, such information is either restructured or eliminated in LLVM IR, making it hard to detect. Source-code level identification of such information is also imprecise and offers limited benefits. PanicKiller performs a cross-IR analysis to extract the semantic information that can only be precisely identified in Rust-specific IRs, and propagates the information to and annotates all corresponding LLVM IR instructions with different metadata. In the following, we discuss the analysis in detail.

Specifically, MIR [25] is a control-flow graph-based representation consisting of basic blocks, each of which contains a sequence of statements ending with a terminator, whereas LLVM IR is a Static Single Assignment (SSA) based representation [21]. Rust runtime safety checks are inserted at the MIR through various terminator instructions [39] based on different categories as summarized in Table 2. Except for customized safety checks, most checks are not visible at the source-code level. After extensive optimization and refactoring during the code generation process, they are expanded into multiple instructions in LLVM IR. As illustrated in Table 3, it is rather challenging to directly identify arithmetic overflow checks, bounds checks, divide/remainder by zero checks, `unwrap` checks, and customized checks in the LLVM IR, due to the introduction of numerous intermediate values and the expansion into multiple new instructions. Therefore, during the code generation phase [7], we identify runtime safety checks at the MIR and annotate the corresponding LLVM IR instructions with metadata.

Similarly, during the translation from MIR to LLVM IR, the unsafe information is optimized out. Unsafe Rust includes not only code blocks and functions wrapped by `unsafe` keywords, but also raw pointer usage, which can appear in both safe and unsafe code. Therefore, source-code level analysis like `unsafe` keyword identification is imprecise and insufficient. To facilitate the identification of unsafe code in LLVM IR, we leverage the annotation strategies from existing work [64]. In particular, we locate the raw pointer type, as well as unsafe code blocks and functions marked by `unsafe` keywords at the Rust MIR and HIR [14], and propagate this information to LLVM IR through metadata. Furthermore, PanicKiller conducts points-to and value-flow analyses [68] on raw pointer types in LLVM IR to detect unsafe memory accesses, thereby achieving a more comprehensive identification of unsafe code.

## 5.3 Dynamic Taint Analysis

We have identified that different execution paths of Rust programs would contain numerous and diverse categories of runtime safety checks, and these checks would enforce different safety conditions. To pass these runtime safety checks, certain bytes in the input need to satisfy the corresponding safety conditions, such as correct value ranges, non-zero or non-null values, and value bounds implied by Rust types, *etc.* To identify these critical bytes, we propose a novel dynamic taint analysis technique to generate high-quality inputs that are more likely to pass the runtime safety checks. The dynamic taint analysis in PanicKiller tracks the critical input bytes that are associated with the safety conditions. Besides, it analyzes the variables in the safety conditions, especially by inferring their types.

**Table 3: Source code examples related to Rust runtime safety checks, and the corresponding simplified LLVM IR instructions. The actual number of instructions is much higher.**

| Code examples | Inserted checks in simplified LLVM IR |
|---|---|
| `u8:MAX + 1;` | `%r, %f = call @llvm.uadd.with.overflow.i8(1, %v)`<br>`br %f, label %panic, label %bb`<br>`panic: invoke core::panicking::panic("overflow..",)` |
| `array[len + 1..];`<br>`&s[1..]; //when s="™"` | `%r = getelementptr inbounds {}, {}* %slice`<br>`%f = icmp ult i64 %v, %slice.1`<br>`br %f, label %panic, label %bb`<br>`panic: call core::panicking::panic_bounds_check` |
| `a / b; //when b = 0` | `%f = icmp eq i8 %denominator, 0`<br>`br %f, label %panic, label %bb`<br>`panic: invoke core::panicking::panic("divide..",)` |
| `x.unwrap(); //when x = None/Err` | `%r = load {}*, { %self }** %option`<br>`%f = icmp eq %r, null`<br>`switch %f, label %panic, label % bb`<br>`panic: call core::panicking::panic("unwrap None..",)` |
| `panic!("A panic!");`<br>`assert!(1 == 0);` | `%f = icmp / fcmp...`<br>`br %f, label %panic, label % bb`<br>`panic: call core::panicking::panic_fmt("A panic!")` |

This helps reveal value ranges imposed by Rust types, which can mitigate overflow issues. Furthermore, PanicKiller introduces new mutation strategies to improve the likelihood of passing runtime safety checks during fuzzing. In the next, we present the details of critical byte tracking, type inference, and mutation strategies of PanicKiller's dynamic taint analysis, respectively.

*5.3.1 Critical Byte Tracking.* The safety-relevant variables (safety variables in short) in the safety conditions could be influenced by only certain bytes in the input. Our objective is to identify and track these critical bytes, and mutate them to pass the safety conditions, rather than mutating the entire input, which would be unproductive. This requires PanicKiller to perform taint tracking on the critical bytes that reach the safety variables in the safety check conditions. However, unlike previous work [49, 62] that mainly focuses on the constraints imposed by classic conditional statements, runtime safety checks enforce more complex conditional statements, as illustrated in Table 3. Consequently, we design new critical byte tracking strategies tailored to different types of runtime safety checks, and discuss the details as follows.

**Safety variable identification.** Specifically, the cross-IR analysis (§5.2) in PanicKiller annotates various LLVM IR instructions corresponding to runtime safety checks. Thus, we only need to mark the relevant safety variables within these annotated instructions as sinks and perform forward taint analysis from the program inputs as sources to identify the critical input bytes. In contrast to earlier taint analysis work [49, 62], which concentrate on a limited set of instructions (*e.g.*, `cmp` related instructions), we precisely identify different types of safety variables from different safety conditions in more complex instructions.

As shown in Table 3, for algorithmic overflow checks, the `call` instruction invokes LLVM intrinsic functions such as `llvm.sadd/uadd`, `ssub/usub`, and `smul/umul.overflow.*` [5] to perform overflow checks. We identify the arguments of these function calls as safety variables for tracking. Regarding bounds checks, the `getelementptr` instruction is used to obtain the address of

a subelement of an aggregate data structure (*e.g.*, array, slice, and algebraic data types in Rust). It computes and returns both the address and its offset index of the base address. However, the returned address is not necessarily within the bounds of the allocated object. The `icmp` instruction then compares whether the index is out of bounds. We consider the index argument of the `icmp` instruction as a safety variable. Similarly, for divide- or remainder-by-zero checks, we identify the `denominator` operand in the `icmp` instruction as a safety variable.

The logic for `unwrap` checks is more complex. Essentially, a `load` instruction would unwrap an `Option` or `Result` type variable to retrieve its internal value. If an `icmp` instruction detects that the value is null, it would trigger a panic. Therefore, we treat the argument of the `icmp` instruction, which is also the result value returned by the `load` instruction, as a safety variable to track whether any input byte would affect the value of an `Option` or `Result` type. If such bytes exist, their values should not be null; otherwise, it would fail `unwrap` safety checks. For customized checks, we identify safety variables of the conditional instructions that may lead to the basic blocks corresponding to customized panic checks. We track the conditional instructions, such as `icmp`, `fcmp`, `cmpxchg` intrinsics, as well as common calls [62] like `strcmp`, `memcmp`, *etc.*

After initially identifying the safety variable candidates based on the characteristics of different safety conditions, we perform further validation to filter out irrelevant safety variables. It is observed that in the logic of runtime safety checks, there are panic basic blocks that execute error handling, such as printing error messages and aborting the program. Thus, we locate the basic blocks of the identified safety variables, and assess whether their successors include panic basic blocks through control flow instructions (*e.g.*, `br`, `switch`, *etc.*). Subsequently, we employ the symbolic matching approach to determine whether the panic basic blocks contain instructions responsible for handling the runtime safety check failures. We utilize error-handling keywords (*e.g.*, overflow, panic_bounds_check, divide, unwrap, panic_fmt, *etc.* as shown in Table 3) for each category of runtime safety checks. If the validation succeeds, it demonstrates that we accurately identify the safety variables that are related to the safety conditions.

**Critical byte tracking.** After identifying the safety variables, PanicKiller performs byte-level forward taint analysis from the input sources to track how each input byte propagates through program execution. PanicKiller monitors taint propagation and records the set of critical bytes that reach the safety variables in the runtime safety checks. In most cases, we can collect a determined number of input bytes and include them as critical bytes. Nevertheless, PanicKiller skips the safety variables whose values are not derived from program inputs. As demonstrated in the example in Listing 1, the bytes of the `usize` type `val` are identified as critical, as they are associated with multiple safety conditions enforced by the four runtime safety checks at lines 3-11. Our critical byte tracking captures the relationships between the critical input bytes and all the safety variables they influence. It also indicates that slight changes in these critical bytes may have an impact on numerous safety conditions.

*5.3.2 Type Inference.* As Rust is a strongly typed language that enforces type checking on values, inferring the types of safety variables allows us to understand the corresponding safety conditions, such as the bounded integer ranges imposed by Rust types. The type information can help subsequent input mutation to satisfy the safety conditions and mitigate the overflow issues. We discuss how we achieve precise type inference in PanicKiller. Our critical byte tracking in §5.3.1 is designed upon existing LLVM-based dynamic taint analysis engines [18, 20, 49, 55]. Although these engines can identify input offsets corresponding to the critical bytes that are associated with the safety conditions, as well as determine the size of these bytes which are used collectively as a single value, we identify several limitations of their type inference capabilities.

Their type inference may result in imprecision, and is not well-suited to the Rust language. First, they infer types based on the size of the critical bytes processed by the programs. For instance, if a C program processes a 4-byte data in the input, it leads to an inference of an `int32` type. However, due to the discrepancies between the Rust and C/C++ type systems, this method results in considerable inaccuracies. For example, an 8-byte data in Rust can represent a `u64`, `i64`, or `usize` type. While the `bool` type in C is represented as an `i1` type in LLVM IR, Rust's `bool` type is usually represented as an `i8` type [8], making it difficult to accurately infer whether a 1-byte data in Rust represents `u8`, `i8`, or `bool` type. Second, parts of the type inference capabilities of the current engines are limited to some specific instructions, such as by hooking `libc` library calls where detailed type information for the arguments is available. For instance, they would infer `char*` from `strlen`, `strnlen`, or `strdup`, *etc.* This rule-based type inference approach does not apply to our Rust scenario, given that the safety conditions we track are considerably more complex as discussed in §5.3.1.

To address these limitations, we propose a new and effective type inference approach by leveraging the Rust type system. To precisely infer the types of safety variables related to runtime safety checks in LLVM IR, we extend the cross-IR analysis in §5.2. We collect the types that the Rust compiler infers during the MIR phase, and propagate them to the corresponding variables in LLVM IR. Specifically, we identify the safety variables of instructions related to runtime safety checks in the MIR. By utilizing the TyKind [43] API in MIR, we can obtain the types inferred by the compiler during compilation. Then when the MIR variables are transformed into LLVM IR variables during the code generation phase, we modify the translation process and propagate the inferred types via metadata [45]. This enables us to directly access the types of safety variables at the LLVM IR instructions, facilitating precise type inference. Note that our approach is limited to inferring primitive types (*e.g.*, `int`, `float`, `bool`, *etc.*) in Rust. In fact, the primitive type inference applies to the safety variables of many runtime safety checks. We consider the type inference for complex types such as algebraic data types, generic types, *etc.* as future work. Even if PanicKiller fails to infer the types of some safety variables, this does not significantly prevent subsequent mutation from finding a solution to pass the runtime safety checks.

*5.3.3 Mutation Strategies.* PanicKiller has determined the critical input bytes and inferred their types. We need to decide how to mutate them intelligently to pass various runtime safety checks. Using

random mutation is unlikely to find satisfactory values efficiently and is prone to violating safety conditions. Using the gradient descent algorithm [49] or other heuristic-based mutation to pass the constraints would be complex. For example, in the gradient descent approach, each safety condition is modeled as a black-box function to be solved. Given the potentially large number of runtime safety checks, it is difficult and impractical to solve these functions simultaneously.

We observe that a common pattern is the monotonic correlation between the critical bytes and the safety variables, such as the overflow bounds and array index bounds. To pass the safety conditions and explore more code blocks, we adopt a linear mutation strategy. We apply the linear mutation method by adding or subtracting a value (*e.g.*, one) to or from the critical bytes. As it is difficult to infer the logical relationships between the bytes and variables, and a set of critical bytes may be associated with multiple safety variables, identifying the mutation directions is very challenging. We initially apply linear increasing mutation. If this direction leads to repeated crashes caused by failed runtime safety checks, PanicKiller would switch to linear decreasing mutation to reduce such failures and improve input quality. Most runtime safety checks enforce upper bounds on the critical bytes, such as algorithmic overflow checks and bounds checks. Since we have inferred the types of these critical bytes, we constrain their maximum values (*e.g.*, `u8:MAX`, *etc.*) during the mutation to mitigate overflows. For divide/remainder-by-zero and `unwrap` checks, we set the corresponding bytes as non-zero or non-null, respectively. For customized checks, we continue to employ the linear mutation method, which prevents random perturbations of the bytes from producing abnormal values. Although this approach might still violate some safety conditions and lead to program panics, it could help pass many runtime safety checks and cover a broader range of code. For the remaining bytes in the input, we employ random exploration mutations, such as bit flipping, byte flipping, arithmetic operations, havoc [51], *etc.* to help discover more uncovered code.

## 5.4 Input Prioritization

After we design how to mutate the input bytes, PanicKiller performs input prioritization to select promising inputs for further exploration and mutation, thereby efficiently allocating fuzzing energy. To pass runtime safety checks and uncover more bugs that mainly stem from unsafe code, PanicKiller performs input prioritization to exercise program paths containing more unsafe code and fewer runtime safety checks. Specifically, first, since unsafe code in Rust is the main cause of memory issues in Rust, the inputs should cover the unsafe code with the highest priority. Second, to reduce the difficulty and overhead incurred by passing the safety conditions, we also give higher priorities to paths with fewer runtime safety checks, among all paths that contain the runtime safety checks. In real-world Rust code, most of the unsafe code is not directly exposed [54]. Unsafe code is typically encapsulated intra-procedurally by safe code or is called inter-procedurally by safe functions. To reach the unsafe code, in most cases, the fuzzer needs to go through the safe code and pass the runtime safety checks. Therefore, these input prioritization strategies in PanicKiller can complement each other.

We propose new coverage accounting metrics in PanicKiller for input prioritization. The metrics measure the number of runtime safety checks and unsafe code along edges between basic blocks, and we use them as feedback during fuzzing. First, we assess an edge by the number of unsafe related operations involved in the destination basic block, which includes whether the basic block itself is unsafe and the number of unsafe functions called within this block. Formally, let $dst_e$ denote the destination basic block of edge $e$, let $unsafe(b)$ denote whether basic block $b$ is unsafe, and let $num_u$ denote the number of called unsafe functions. For an $e$, we have:

$$Unsafe(e) = unsafe(dst_e) + num_u(dst_e)$$

where $Unsafe(e)$ represents the unsafe metric. Second, another metric is the number of runtime safety checks inserted in the destination basic block followed by the edge. In most cases, there may exist only one runtime safety check inserted (*e.g.*, at the end) in a basic block, or none at all. In other cases, there may be multiple due to compiler optimizations. For execution paths containing different numbers of runtime safety checks, PanicKiller prioritizes in those with fewer checks. For paths without any runtime safety check, they are given the lowest priority. Formally, we let $num_p$ denote the number of runtime safety checks in the destination basic block. For an $e$, we have the panic metric:

$$Panic(e) = num_p(dst_e)$$

We propose the following rules for input prioritization. When an input has a higher unsafe metric value, it is preferred. If the unsafe metric values are equal, we consider inputs with non-zero panic metric values, and prefer the one with lower panic metric value. PanicKiller selects the relevant test cases that either cover new edges, or increase the hit count of an existing edge to a new scale as interesting, while recording their metric. It then prioritizes in the test cases that perform better based on our comparison. PanicKiller adopts the classic power scheduling method in AFL++ [51]. In particular, it first assigns more energy to inputs that exercise paths with more unsafe code and fewer runtime safety checks. The other inputs with suboptimal metric will be explored and covered later.

## 5.5 Fuzzing Loop

Algorithm 1 illustrates the entire process of PanicKiller's fuzzing loop. The algorithm accepts a set of seed inputs $I$, two versions of the target programs (including one instrumented version $program_t$ for taint tracking and one without $program_{nt}$ for efficiency), as well as the unsafe metric and panic metric for input prioritization. As an initial step, PanicKiller identifies the critical bytes in the inputs that are associated with safety variables enforced by runtime safety checks, along with their inferred types (line 1). These seed inputs are then used to initialize the fuzzing queue (line 2). During each iteration of the fuzzing loop (lines 3–17), an input is prioritized based on the unsafe and panic metrics (line 4). The selected input undergoes our linear and random mutation strategies to pass runtime safety checks (line 5). If the execution leads to a crash, PanicKiller determines whether it is caused by the runtime safety check or a potential bug, and the corresponding input is recorded (lines 6–9). Besides, if the execution trace indicates the input is

**Algorithm 1** PanicKiller's fuzzing loop.

**Input:** $I$: seed inputs, $program_t$: program with taint instrumentation, $program_{nt}$: program without taint instrumentation, $m_u$: unsafe metric, $m_p$: panic metric.
**Output:** $C$: crashing inputs.

1: $cstrs, I' \leftarrow initializeSeeds(program_t, I)$ // Unexplored safety conditions and the updated seed inputs, where critical bytes associated with the conditions are identified.
2: $Q \leftarrow I'$ // Initialize queue with updated seed inputs.
3: **while** True **do**
4: $i \leftarrow selectInput(Q, m_u, m_p)$ // Input prioritization based on the unsafe and panic metric.
5: $i' \leftarrow mutate(i)$ // Linear and random mutation.
6: $result, trace \leftarrow program_{nt}(i')$
7: **if** $result == crash$ **then**
8: $C \leftarrow C \cup \{i'\}$
9: **end if**
10: **if** $isInteresting(trace)$ **then**
11: $i'' \leftarrow program_t(i')$ // Critical byte tracking.
12: $Q \leftarrow Q \cup \{i''\}$ // Add to queue if new coverage found.
13: **end if**
14: **if** $timeout$ or $max_execs$ reached **then**
15: **break**
16: **end if**
17: **end while**

interesting, PanicKiller tracks its critical bytes and re-enqueues the input for further exploration (lines 10–13). The loop continues until a predefined timeout or a maximum number of executions is reached (lines 14–17).

## 6 Implementation

We implemented a prototype of PanicKiller with around 2,500 lines of C code and 1,000 lines of Rust code. Next, we discuss the most important implementation details of PanicKiller.

**Cross-IR analysis.** We implemented the automatic metadata annotation of cross-IR analysis by modifying the source codes of the `codegen-ssa` and `codegen-llvm` within `rustc`. For runtime safety check annotation, PanicKiller analyzes all terminator instructions [40] (*e.g.*, `Call`, `Assert`) associated with these checks during the code generation phase. It then injects metadata into the LLVM IR instructions generated from that terminator. Specifically, statement and terminator instructions involve operands, namely `place` that indicates memory location, and `rvalue` that represents value production. The Rust compiler can infer the type (`TyKind` [43]) of the variable in both `place` and `rvalue`. To infer the types of the safety variables, PanicKiller analyzes all corresponding variables from the statements and terminator within the basic block that contains the runtime safety checks, and propagate these inferred types to the LLVM IR instructions. For unsafe code annotation, PanicKiller employs the annotation strategies of ERASan [64].

**Dynamic taint analysis.** Dynamic taint analysis collects all input bytes that affect safety conditions from taint access records. We built our taint analysis on Polytracker [20] instead of Angora [49], libdft64 [18], *etc.* Their taint analyses all output the same influencing bytes for a given safety variable, because they are all built on top of the LLVM Data Flow Sanitizer [11], yet Polytracker offers two advantages. First, whereas the taint analyses in Angora and libdft64 can only track the provenance of up to 16 taints at once, Polytracker can track up to $2^{31}$-1 taints. Second, it can track every input byte at once while imposing negligible performance overhead for almost all inputs. We enable the `lto="thin"` flag in the Rust compiler to link all dependent code of Rust programs, thereby facilitating a comprehensive dynamic taint analysis to cover a broad range of safety conditions.

**Others.** Due to the strict version compatibility restrictions imposed by the Rust compiler [42], we conducted our implementation on `rustc` 1.76.0, LLVM 17.0.6, afl.rs 0.15.9, and AFL++ 4.21c, which are all recent versions. Our input prioritization and mutation mechanisms were implemented as LLVM passes. To enable their functionalities for Rust programs, the `-Z llvm-plugins` flag must be enabled in the Rust compiler.

## 7 Evaluation

We extensively evaluate the performance of PanicKiller by conducting the following experiments. First, we show that PanicKiller helps more effectively and efficiently detect the bugs in a benchmark dataset compared with the state-of-the-art Rust fuzzers in §7.1. We then conduct a component-wise analysis to explain the effectiveness of our design choices in §7.2. Finally, we use PanicKiller for unknown bug detection in §7.3.

**Experiment setup.** We conduct experiments on an x86_64 server with a 28-core 2.10 GHz Intel Xeon Platinum 8450H CPU and 1 TB of memory running Ubuntu 22.04.5 LTS.

### 7.1 Benchmark Evaluation

We selected the real-world benchmark dataset composed of Rust packages with bugs as introduced early in §3 for our evaluation. The selected 16 packages in our benchmark dataset have all been extensively evaluated in recent Rust library fuzzing research [56, 71, 72], and they are representative. Rust packages are developed for two main purposes [27]. One is to serve as a binary that creates an executable program. The other is to act as a library defining functionalities intended for sharing among multiple programs. Rust libraries, also known as Rust crates, account for a large proportion, with about 176,500 libraries collected on `crates.io` [31]. Although we could incorporate more new packages such as binary packages into our benchmark dataset, it would be unnecessary and unfair for comparison as previous works did not evaluate other Rust packages. To reflect the essence of each bug, we switched all the corresponding packages to the versions where the bugs occurred.

We choose current state-of-the-art Rust fuzzers, namely afl.rs [13], cargo-fuzz [30], and honggfuz.rs [12], as well as PanicKiller to conduct comparison. These three Rust fuzzers integrate AFL++ [51], libFuzzer [19], and honggfuzz [15] respectively, with adaptations for testing Rust programs. For the sake of fairness, we exclude other LLVM-based coverage-guided and directed fuzzers [63]. The reasons are twofold. First, other LLVM-based coverage-guided fuzzers (mostly C/C++ fuzzers) are unaware of Rust features like runtime safety checks and the Rust type system. Second, existing directed

**Table 4: Comparison results on the benchmark dataset. Parts of the results from afl.rs, cargo-fuzz, and honggfuzz.rs are presented early in Table 1. TTE denotes the mean time-to-exposure calculated among all the detected bugs. Cov represents the edge coverage. Checks denote the statically computed average number of runtime safety checks along all paths to all the detected bugs. + number, × ratio, and ÷ ratio represent the average improvement of PanicKiller compared to the three fuzzers, which exhibit similar performance.**

| Package | afl.rs | | | cargo-fuzz | | | honggfuzz.rs | | | PanicKiller | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TTE | p | Cov | TTE | p | Cov | TTE | p | Cov | TTE | Cov | Crashes | Bugs | Checks |
| autocfg | 306 | 0.011 | 110 | 434 | 0.008 | 102 | 384 | 0.005 | 117 | 34 (÷11.0) | 165 (×1.5) | 2 (÷4.3) | 2 (+1) | 3 |
| bumpalo | 2268 | 0.002 | 284 | 2676 | 0.005 | 260 | 2809 | 0.003 | 289 | 263 (÷9.8) | 532 (×1.9) | 6 (÷3.9) | 2 | 8 |
| byteorder | 34877 | 0.005 | 76 | 48083 | 0.003 | 70 | 42445 | 0.009 | 75 | 8264 (÷5.0) | 133 (×1.8) | 23 (÷13.5) | 6 | 12 |
| bytes | 9595 | 0.006 | 387 | 10477 | 0.011 | 360 | 8926 | 0.006 | 379 | 2447 (÷3.9) | 688 (×1.8) | 20 (÷13.3) | 6 (+2) | 18 |
| clap | 52509 | 0.004 | 8054 | 51336 | 0.005 | 7323 | 58848 | 0.004 | 7859 | 9253 (÷5.8) | 16464 (×2.1) | 39 (÷31.3) | 3 (+1) | 90 |
| fixedbitset | 3572 | 0.007 | 336 | 2900 | 0.003 | 343 | 4323 | 0.009 | 331 | 1444 (÷2.4) | 588 (×1.7) | 22 (÷2.9) | 2 | 10 |
| flate2 | 1720 | 0.004 | 136 | 1878 | 0.008 | 127 | 1800 | 0.007 | 131 | 171 (÷10.5) | 207 (×1.5) | 2 (÷6.0) | 1 | 2 |
| mio | 496 | 0.008 | 60 | 541 | 0.002 | 56 | 561 | 0.004 | 59 | 65 (÷8.1) | 96 (×1.6) | 1 (÷6.6) | 1 | 4 |
| proc-macro2 | 2187 | 0.004 | 1176 | 2688 | 0.003 | 1066 | 3034 | 0.008 | 1138 | 239 (÷11.0) | 1960 (×1.7) | 6 (÷19.5) | 1 | 30 |
| regex | 47628 | 0.009 | 23840 | 52006 | 0.002 | 21612 | 54307 | 0.007 | 23451 | 13231 (÷3.8) | 34250 (×1.5) | 80 (÷64.8) | 5 (+3) | 192 |
| csv | 2927 | 0.005 | 2170 | 3296 | 0.006 | 1971 | 3122 | 0.003 | 2120 | 344 (÷9.0) | 3761 (×1.8) | 21 (÷6.2) | 2 | 26 |
| url | 30431 | 0.003 | 6536 | 33228 | 0.002 | 6008 | 34309 | 0.011 | 6338 | 252 (÷129.5) | 8612 (×1.3) | 6 (÷74.5) | 2 | 40 |
| ryu | 3327 | 0.004 | 64 | 4200 | 0.004 | 59 | 4095 | 0.005 | 62 | 71 (÷54.5) | 102 (×1.6) | 6 (÷6.1) | 6 (+5) | 20 |
| time | 16618 | 0.005 | 1533 | 20145 | 0.009 | 1420 | 17459 | 0.002 | 1511 | 330 (÷54.7) | 3184 (×2.1) | 14 (÷25.1) | 6 (+1) | 32 |
| uni-seg | 2939 | 0.005 | 128 | 3209 | 0.005 | 116 | 3534 | 0.008 | 124 | 148 (÷21.8) | 190 (×1.5) | 11 (÷5.3) | 10 | 6 |
| uuid | 4560 | 0.002 | 206 | 4979 | 0.006 | 189 | 5242 | 0.004 | 203 | 420 (÷11.7) | 297 (×1.4) | 8 (÷8.0) | 2 (+1) | 20 |

fuzzers are not well-suited for detecting deep Rust bugs, making the comparison unfair. Detecting deep bugs requires targeting hundreds of unsafe code locations. This disperses fuzzing energy and makes it difficult to efficiently reach the many targets. Current directed fuzzers like CAFL [58] and GREYONE [52], *etc.* cannot handle such multi-target problem well. Since adapting existing LLVM-based coverage-guided and directed fuzzers to Rust requires significant work and has not been achieved yet, we consider it as future work and discuss more details in §8.

We use RPG [71] to generate identical fuzz drivers for each fuzzer to test the packages. For a fair comparison, we apply the same seeds per package across different fuzzers, and follow the common traditional input generation method [63]. We assign each fuzzer to a dedicated core and leave 20% of the available cores unused to minimize interference. We enable the Address Sanitizer [67] for each fuzzing campaign. We apply AFL-CMin [4] to the reported crashes and present the deduplicated results. As a standard setting in evaluating fuzzers, each fuzzer would test each package with a time budget of 24 hours, and we repeated the testing process five times. We discuss the evaluation results and the overhead of PanicKiller as follows.

**Evaluation results.** The comparison results are shown in Table 4. *PanicKiller outperformed all other fuzzers in detecting bugs in the benchmark dataset.* For the 44 known bugs in the benchmark dataset, afl.rs, cargo-fuzz, honggfuzz.rs, and PanicKiller can detect all of them within the time budget. PanicKiller additionally detected 14 new bugs that the other fuzzers cannot detect. The reason why afl.rs, cargo-fuzz, and honggfuzz.rs can all detect the known bugs is that these bugs were initially identified by these Rust fuzzers. The 14 new vulnerabilities detected by PanicKiller include five dynamic stack buffer overflows, seven segmentation faults on unknown (*e.g.*, high-value) addresses, and two bugs of stack use after scope, which can pose severe security risks. All of these bugs have been confirmed by the developers as true bugs. The detection of these new bugs confirms our hypothesis that long-lasting bugs may reside in deep code locations that are hard to test. However, since the tested Rust package versions are outdated for approximately four years, the corresponding developers do not plan to fix these issues.

As shown in Table 4, we carefully calculated the time required to expose each bug and present the average time needed for revealing all bugs in each package. PanicKiller significantly enhanced bug exposure speed, achieving an average speedup of over 8× in 11 (68.8%) packages and a maximum improvement of up to 129.5× compared to afl.rs, cargo-fuzz, and honggfuzz.rs. We computed the p-value using the Mann–Whitney U test [57] on the time-to-exposure to assess the statistical significance of our experiment results. In all cases, the results were statistically significant with $p < 0.05$, indicating that the influence of random variations in bug exposure time was limited. In terms of coverage, PanicKiller outperformed the other fuzzers by delivering an average improvement of more than 1.5× in 14 (87.5%) packages, with peak improvements reaching 2.1×. Moreover, PanicKiller substantially reduced the number of false-positive crashes, exhibiting an average reduction of over 5× in 13 (81.3%) packages and up to 64.8× improvement compared with the other three fuzzers.

We affirm that the significant improvements of PanicKiller over other fuzzers are due to our new design. It enables the fuzzer to pass runtime safety checks, while allocating more energy towards detecting bugs in deep and unsafe code. We further discuss the contributions of different components of PanicKiller in §7.2. The experiment results also verify the limitations of current fuzzers we summarize in §3.1. Current Rust fuzzers exhibit low effectiveness and efficiency in bug detection, as they tend to trigger a large number of false-positive crashes and devote substantial energy to exercising only shallow code, leading to early termination of the programs. Specifically, the input mutation capabilities of these fuzzers would easily lead to the generation of (large) values that fail the runtime safety checks, and the subsequent fuzzing based on these inputs would not make much progress. Our benchmark evaluation demonstrates that the current Rust fuzzers which are unaware of runtime safety checks, are not well-suited for Rust fuzzing.

**Table 5: The component-wise evaluation results of PanicKiller. The experiment results of afl.rs and PanicKiller itself are presented in Table 4. C/B represents the number of crashes/bugs. Checks represent the average number of runtime safety checks along all paths towards the bugs.**

| Package | afl.rs+unsafe | | | afl.rs+panic | | | afl.rs+dta | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | TTE | Cov | C/B | TTE | Cov | C/B | TTE | Cov | C/B | Checks |
| autocfg | 234 | 126 | 6/1 | 151 | 142 | 6/1 | 92 | 159 | 6/2 | 2 |
| bumpalo | 1743 | 320 | 20/2 | 907 | 344 | 16/2 | 300 | 439 | 8/2 | 8 |
| byteorder | 26933 | 89 | 242/6 | 17043 | 92 | 171/6 | 12358 | 117 | 63/6 | 12 |
| bytes | 7432 | 454 | 200/4 | 5917 | 484 | 175/4 | 3557 | 587 | 54/6 | 18 |
| clap | 39700 | 9544 | 812/2 | 24506 | 9702 | 570/2 | 15809 | 12672 | 260/2 | 86 |
| fixedbitset | 2989 | 377 | 48/2 | 2830 | 432 | 40/2 | 1650 | 517 | 38/2 | 10 |
| flate2 | 1349 | 150 | 11/1 | 833 | 167 | 10/1 | 778 | 194 | 6/1 | 2 |
| mio | 391 | 71 | 5/1 | 247 | 77 | 4/1 | 111 | 85 | 4/1 | 4 |
| proc-macro2 | 1246 | 1326 | 86/1 | 994 | 1495 | 68/1 | 414 | 1857 | 18/1 | 30 |
| regex | 45774 | 28253 | 4207/2 | 40617 | 28894 | 2709/2 | 19434 | 30232 | 162/2 | 184 |
| csv | 2236 | 2405 | 104/2 | 1331 | 2704 | 60/2 | 418 | 3263 | 26/2 | 26 |
| url | 13227 | 7738 | 341/2 | 8579 | 7963 | 240/2 | 462 | 8257 | 14/2 | 40 |
| ryu | 1612 | 70 | 25/1 | 1259 | 79 | 21/1 | 268 | 99 | 10/6 | 20 |
| time | 11517 | 1723 | 264/5 | 7733 | 1860 | 186/5 | 814 | 2384 | 80/5 | 29 |
| uni-seg | 1325 | 147 | 38/10 | 940 | 160 | 30/10 | 356 | 175 | 20/10 | 6 |
| uuid | 3431 | 242 | 49/1 | 1890 | 255 | 42/1 | 878 | 276 | 22/2 | 20 |

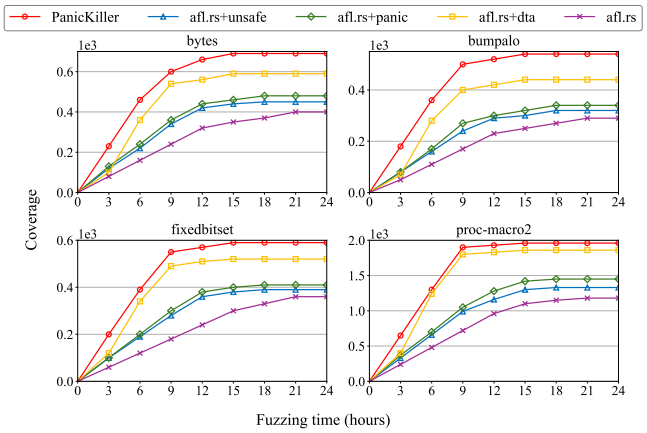


**Figure 2: Coverage for different fuzzers in different packages.**

**Overhead.** We measured the compile-time and runtime overhead incurred by PanicKiller for each package in the benchmark dataset. We computed the average compile-time overhead by comparing the time required to build each Rust package using PanicKiller to that using the baseline afl.rs. PanicKiller took 82,045 microseconds on average (6.1× overhead), whereas PanicKiller excluding taint instrumentation (static-analysis only) took 53,800 microseconds (4.0× overhead). Nevertheless, PanicKiller's compile-time analyses require only one-time costs. Besides, our dynamic taint analysis is built on PolyTracker [20], which imposes modest or even negligible runtime overhead for all inputs. Given that the overhead of PanicKiller is significantly less than the fuzz testing time, we consider it negligible.

## 7.2 Component-wise Analysis

We aim to evaluate our contributions in prioritizing in the testing of unsafe code, the exploration of paths with fewer runtime safety checks, and dynamic taint analysis to pass safety conditions. We extend the baseline fuzzer afl.rs with our new design to conduct component-wise analyses to demonstrate the effectiveness and advantages of each component of PanicKiller. To evaluate our unsafe metric of input prioritization, we modify afl.rs by replacing its coverage accounting metric with ours; we refer to this setup as afl.rs+unsafe. Similarly, to assess our panic metric of input prioritization, we modify afl.rs with our panic metric, denoting this setup as afl.rs+panic. We evaluate the input prioritization metrics separately because their objectives are essentially different. We represent another setup as afl.rs+dta, which incorporates our dynamic taint analysis mechanism into afl.rs to evaluate its impact. We also used the benchmark dataset as the component-wise evaluation dataset, and reused the same experiment setup in §7.1. We ran each tool for 24 hours with five repetitions. The results of these three fuzzers, as well as afl.rs and PanicKiller, are shown in Table 5 and Table 4, respectively.

We computed the overall average improvement across all packages for these fuzzers compared to afl.rs, to demonstrate the effectiveness of different components of PanicKiller. Specifically, afl.rs+unsafe enhanced bug exposure speed, improved coverage, and reduced crashes, with average improvements of 1.4×, 1.14×, and 1.2× across all packages. Afl.rs+panic improved bug exposure speed, coverage, and crash reduction by averages of 2.1×, 1.24×, and 1.5× among all packages, respectively compared to afl.rs.

The experiment results indicate that both our unsafe metric and panic metric in input prioritization are effective, and are superior to the baseline. The main reason is that afl.rs tends to get trapped in code containing lots of runtime safety checks, leading to the early termination of the programs. Consequently, deep code remained unexplored, resulting in limited coverage. Our unsafe metric and panic metric can alleviate this issue. This is because most execution paths would contain runtime safety checks, and these two metrics can help prioritize in those with fewer runtime safety checks. Specifically, unsafe code in Rust inherently has few or even no runtime safety checks, thus the unsafe metric essentially helps avoid these checks and exercise code that is more likely to trigger memory bugs. Note that the panic metric exhibits better performance than the unsafe metric. This is because in most cases, reaching unsafe code still requires going through safe code and passing the runtime safety checks. Therefore, by design, the panic metric prefers paths with fewer runtime safety checks and achieves superior results.

We further study the effectiveness of dynamic taint analysis. Afl.rs+dta enhanced bug exposure speed, improved coverage, and reduced crashes, with average improvements of 9.7×, 1.46×, and 6.5× respectively among all packages compared to afl.rs. Due to our novel design, dynamic taint analysis achieved superior performance which was even better than the results of input prioritization. This substantiates our observation that passing runtime safety checks enables deeper code exploration. Moreover, we measured that PanicKiller achieved average improvements of 20.1×, 1.66×, and 16.7× respectively for bug exposure speed, coverage, and crash reduction compared to afl.rs. The results demonstrate that PanicKiller with all components integrated, performed the best.

We further present the process of increasing coverage for each fuzzer in Figure 2, as code coverage is an important measure. Given that we observed most Rust packages exhibited similar trends, we selected the bytes, bumpalo, fixedbitset, and proc-macro2 packages as the examples. As the figure illustrates, afl.rs+unsafe and afl.rs+panic were able to increase coverage at an early stage, but

they had difficulties exercising paths with numerous runtime safety checks. In contrast, afl.rs+dta initially improved coverage at a slower speed than afl.rs+panic because it did not prioritize in paths with fewer safety conditions. However, due to its special capability of passing safety conditions, it gradually increased its coverage until it reached deeper code. This demonstrates the importance of integrating input prioritization with dynamic taint analysis in Rust fuzzing, which can substantially help pass runtime safety checks and detect bugs in deep and unsafe code.

## 7.3 Unknown Bug Detection

We further applied PanicKiller to detect unknown bugs in the real world for its superb fuzzing performance. Our experiments demonstrate that PanicKiller is capable of identifying many unknown vulnerabilities in well-maintained Rust packages. We randomly included 30 Rust packages from the fuzzing trophy case [10] for testing. The selected packages have been reported and assigned with RustSec advisory IDs [36] in the past, which indicates the potential for more vulnerabilities. These packages possess high code quality and are frequently updated or maintained; accordingly, we switched them to their latest versions for fuzzing. Each package was fuzzed for 24 hours.

PanicKiller identified 53 new vulnerabilities in 22 Rust packages. Since we conducted the fuzzing in parallel, the experiment was completed in approximately 72 hours. We responsibly reported all the new vulnerabilities to the relevant developers. At the time of writing, 38 vulnerabilities have been confirmed, five vulnerabilities have been fixed, and 11 RustSec IDs have been assigned. The complete details of all detected vulnerabilities will be publicly available along with our artifact. The main reason why some bugs have not been confirmed in a timely manner is that we are still awaiting further responses from the developers. We are also in the process of reporting the confirmed vulnerabilities to RustSec advisories [36] and the CVE program.

The results in Table 6 show that PanicKiller was able to identify various types of vulnerabilities, including 24 stack and heap overflows, one bug of stack use after scope, eight segmentation faults on unknown addresses, three bugs of invalid allocation alignment, and several program aborts, *etc.* It also demonstrates that PanicKiller can discover new vulnerabilities in complex and large-scale software such as Mozilla [26]. Specifically, we further elaborate on the invalid state bugs and abort bugs detected by PanicKiller. These bugs mainly stemmed from the inadequate sanitization and checks within the Rust packages. This would cause the programs to receive anomalous values, which in turn leads to errors such as allocation-size-too-big errors, calloc-overflow errors [3], or running into invalid states, *etc.*, resulting in program aborts. These errors were not handled by Rust's error-handling mechanisms like runtime safety checks, thereby posing severe security risks.

We present a case study in Listing 2 to illustrate a new bug PanicKiller detected in the grcov package. This Rust code snippet contains several runtime safety checks, such as an `unwrap` check at line 4 and a subtraction overflow check at line 10. Bounds checks are also inserted when accessing the `Vec` array. Besides, the execution paths reaching this function contain additional runtime safety checks. However, the use of `unsafe` code at line 10 bypasses

**Table 6: Real-world vulnerabilities found by PanicKiller. Checks represent the average number of runtime safety checks in all paths towards the bugs.**

| Package | Vulnerabilities | Status | Checks |
|---|---|---|---|
| fast-float-rust | 1*SEGV on unknown address | 1*fixed | 8 |
| grcov | 1*SEGV on unknown address | 1*fixed | 24 |
| kalker | 1*SEGV on unknown address | 1*fixed | 26 |
| ouch | 1*SEGV on unknown address | 1*fixed | 12 |
| xmas-elf | 1*stack buffer overflow | 1*fixed | 6 |
| aho-corasick | 2*invalid state | 2*confirmed | 158 |
| chrono | 2*stack buffer overflow<br>3*SEGV on unknown address<br>3*abort | 2*confirmed<br>3*confirmed<br>3*reported | 26 |
| json-rust | 1*invalid allocation alignment<br>5*abort | 1*confirmed<br>5*reported | 8 |
| lewton | 1*abort | 1*reported | 10 |
| lz4_flex | 1*stack buffer overflow<br>3*heap buffer overflow<br>1*abort | 1*confirmed<br>3*confirmed<br>1*reported | 64 |
| Molten | 3*stack buffer overflow<br>1*dynamic stack buffer overflow<br>1*stack use after scope | 3*confirmed<br>1*confirmed<br>1*confirmed | 14 |
| mp3-metadata | 1*stack buffer overflow | 1*confirmed | 4 |
| ntfs | 1*dynamic stack buffer overflow | 1*confirmed | 10 |
| num-bigint | 2*invalid allocation alignment<br>1*SEGV on unknown address | 2*confirmed<br>1*confirmed | 18 |
| pcapng-rs | 1*abort | 1*reported | 2 |
| regex | 3*dynamic stack buffer overflow | 3*confirmed | 164 |
| rust-secp256k1 | 1*dynamic stack buffer overflow | 1*confirmed | 48 |
| rustc-serialize | 3*abort<br>2*stack buffer overflow | 3*reported<br>2*confirmed | 12 |
| swf-parser | 1*stack buffer overflow | 1*confirmed | 20 |
| tinytemplate | 1*stack buffer overflow | 1*confirmed | 6 |
| ttf-parser | 1*stack buffer overflow<br>1*abort | 1*confirmed<br>1*reported | 18 |
| uni-seg | 2*stack buffer overflow | 2*confirmed | 8 |

Rust safety guarantees, and leads to a buffer underflow bug when the `line_num` is zero, causing a segmentation fault (SEGV) on an unknown address reported by Address Sanitizer. PanicKiller successfully detected this bug by passing the runtime safety checks, whereas other fuzzers failed to detect it as they cannot easily pass the checks. The developer acknowledged the implementation issue and fixed the bug.

Furthermore, we employed our static analyzer mentioned in §3.2, to assess the number of runtime safety checks within the packages that had newly detected vulnerabilities. Our results in Table 6 demonstrate that, the detected vulnerabilities are all preceded by a substantial number of runtime safety checks. This finding further verifies our hypothesis that there might exist numerous previously unknown bugs in deep code after passing runtime safety checks. We believe that by expanding the experiment scale, as well as analyzing less-maintained packages, PanicKiller will likely find more vulnerabilities.

# 8 Discussion

**Disabling runtime safety checks.** One may consider disabling all Rust runtime safety checks to eliminate their impacts on fuzzing. However, this approach is almost infeasible. In practice, only small

```rust
// grcov/src/covdir.rs
fn get_coverage(coverage: BTreeMap<u32, u64>) -> (usize, usize, Vec<i64>) {
    let mut covered = 0;
    let last_line = *coverage.keys().last().unwrap_or(&0) as usize;
    let total = coverage.len();
    let mut lines: Vec<i64> = vec![-1; last_line];
    for (line_num, line_count) in coverage.iter() {
        let line_count = *line_count;
        unsafe {
            *lines.get_unchecked_mut((*line_num - 1) as usize) = line_count as i64;
        }
        covered += (line_count > 0) as usize;
    }
    (total, covered, lines)
...
```

**Listing 2: A simplified real-world bug detected in the grcov package.**

portions of these checks can be disabled [16], such as by compiling Rust programs in release mode, or by enabling specific compiler flags (*e.g.*, `-Coverflow-checks=no`). Nevertheless, people usually conduct fuzzing in debug mode. With the runtime safety checks disabled, the data flows leading to crashes can be impractical. This invalidates Rust safety guarantees and largely increases the likelihood of detecting false positive bugs.

**Limitations and future works.** PanicKiller currently supports five categories of runtime safety checks. As discussed in §3, there may exist other types of runtime safety checks, such as those for concurrency and new language features. By following our design, researchers can easily support new runtime safety checks on top of our tool in the future. The design of PanicKiller can be ported to other LLVM-based fuzzers, such as directed fuzzers, to develop new Rust fuzzers. This would first require adapting the LLVM-based fuzzers to support testing Rust programs, which requires a considerable amount of effort. Then after passing runtime safety checks based on our design, the directed fuzzers can reproduce known bugs more rapidly. Besides, treating unsafe code as targets becomes a multi-target directed fuzzing problem. Efficiently reaching hundreds of targets is rather challenging, and we consider it as a future work. Moreover, taint propagation in a complex program with numerous constraints might be time-consuming. Nevertheless, in our evaluation experiments, all taint analyses were completed within a relatively short time. This issue can be further mitigated by monitoring fewer runtime safety checks.

## 9 Related work

**Coverage-guided fuzzing.** Many researchers have paid significant attention to coverage-guided fuzzing [63] to identify bugs. It has been applied to many domains such as kernel [2], binary [60], and different programming languages, including JavaScript, Python, and Rust, *etc.* For example, CollAFL [53], TortoiseFuzz [69], and MUZZ [48] proposed new coverage metrics to guide seed selection and input prioritization to achieve better code coverage. To test Rust programs, afl.rs [13], cargo-fuzz [30], and honggfuzz.rs [15] integrate the most widely used LLVM-based fuzzers, namely AFL++ [51], libFuzzer [19], and honggfuzz [15]. These coverage-guided fuzzers are frequently evaluated on Fuzzbench [63]. Besides, tools such as Rulf [56], RPG [71], and FRIES [72] generate fuzz drivers for testing Rust programs and subsequently invoke these three Rust fuzzers. We identify that the existing fuzzers cannot effectively pass Rust runtime safety checks, and achieve good coverage. We design PanicKiller to address these issues.

**Critical bytes mutation.** Mutating correct bytes that pass complex conditions has been a key problem in improving code coverage. Vuzzer [66] collects magic and immediate values, and mutates critical bytes that are likely to satisfy the conditions. Angora and its variants [18, 20, 49] use taint propagation to solve path constraints without symbolic execution. REDQUEEN [47] solves the magic values and checksums by observing the arguments to function calls and comparing instructions via virtual machine introspection. Lafintel [51] splits multi-byte comparisons into several single-byte comparisons at the compiler level. PATA [62] proposes a path-aware taint analysis to achieve better precision. PanicKiller uses dynamic taint analysis to identify the critical bytes that are associated with safety conditions and help pass runtime safety checks.

## 10 Conclusion

We propose PanicKiller, a new effective Rust fuzzer designed for detecting the deep bugs behind the Rust runtime safety checks. By precisely identifying the critical input bytes that influence the conditions enforced by runtime safety checks, PanicKiller performs novel dynamic taint analysis, input prioritization, and input mutation to pass these checks and explore deep and unsafe code. PanicKiller outperformed current state-of-the-art Rust fuzzers in bug exposure speed, coverage, and false-positive crash reduction in the evaluation. With PanicKiller, we further detected 14 and 53 new vulnerabilities in the benchmark dataset and in the real world, with 11 RustSec IDs assigned.

## 11 Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable suggestions and comments. The work described in this paper was substantially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No.: CUHK 14209323).

## References

[1] 2021. Klee as Rust verification tools. https://github.com/project-oak/rust-verification-tools/.
[2] 2021. Syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.
[3] 2025. AddressSanitizer error examples. https://learn.microsoft.com/en-us/cpp/sanitizers/asan-error-examples.
[4] 2025. Afl-cmin - corpus minimization tool for American Fuzzy Lop (afl). https://github.com/AFLplusplus/AFLplusplus/blob/stable/afl-cmin.
[5] 2025. Arithmetic with Overflow Intrinsics in LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html#arithmetic-with-overflow-intrinsics.
[6] 2025. Asserted messages in the Rustc middle. https://github.com/rust-lang/rust/blob/master/compiler/rustc_middle/messages.ftl.
[7] 2025. Backend Agnostic Codegen in Rust Compiler. https://rustc-dev-guide.rust-lang.org/backend/backend-agnostic.html.
[8] 2025. Bool type translation in rustc_codegen_ssa. https://github.com/rust-lang/rust/blob/master/compiler/rustc_codegen_ssa/src/mir/rvalue.rs#L372.
[9] 2025. The borrow checker in Rust. https://rustc-dev-guide.rust-lang.org/borrow_check.html.
[10] 2025. Collection of bugs uncovered by fuzzing Rust code. https://github.com/rust-fuzz/trophy-case.
[11] 2025. Data flow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html.

[12] 2025. Fuzz your Rust code with Google-developed Honggfuzz. https://github.com/rust-fuzz/honggfuzz-rs.
[13] 2025. Fuzzing Rust code with American Fuzzy Lop. https://github.com/rust-fuzz/afl.rs.
[14] 2025. The High-level Intermediate Representation (HIR). https://rustc-dev-guide.rust-lang.org/hir.html.
[15] 2025. Honggfuzz - Security oriented software fuzzer. https://github.com/google/honggfuzz.
[16] 2025. How to completely disable all Rust safety checks? https://users.rust-lang.org/t/how-to-completely-disable-all-rust-safety-checks/21245.
[17] 2025. Keyword Unsafe in Rust. https://doc.rust-lang.org/std/keyword.unsafe.html.
[18] 2025. libdft: Practical Dynamic Data Flow Tracking. https://github.com/AngoraFuzzer/libdft64.
[19] 2025. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[20] 2025. An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracing. https://github.com/trailofbits/polytracker.
[21] 2025. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.
[22] 2025. Macro assert in Rust. https://doc.rust-lang.org/std/macro.assert.html.
[23] 2025. Macro panic in Rust. https://doc.rust-lang.org/core/macro.panic.html.
[24] 2025. Meet Safe and Unsafe in the Rustonomicon. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html#meet-safe-and-unsafe.
[25] 2025. The Mid-level Intermediate Representation (MIR). https://rustc-dev-guide.rust-lang.org/mir/index.html.
[26] 2025. Mozilla free software company. https://github.com/mozilla.
[27] 2025. Packages and Crates in the Rust Programming Language. https://doc.rust-lang.org/book/ch07-01-packages-and-crates.html.
[28] 2025. Press Release: Future Software Should Be Memory Safe. https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/.
[29] 2025. Rust analysis checks prologue. https://rustc-dev-guide.rust-lang.org/part-4-intro.html.
[30] 2025. Rust bindings and utilities for LLVM's libFuzzer. https://github.com/rust-fuzz/cargo-fuzz.
[31] 2025. The Rust community's crate registry. https://crates.io/.
[32] 2025. The Rust developer's feedback on panic crashes. https://github.com/rust-lang/regex/issues/972#issuecomment-1496012528.
[33] 2025. Rust for Firefox. https://wiki.mozilla.org/Oxidation.
[34] 2025. Rust for Linux. https://github.com/rust-for-linux.
[35] 2025. Rust panic checks in the language items. https://github.com/rust-lang/rust/blob/master/compiler/rustc_hir/src/lang_items.rs#L283.
[36] 2025. The Rust Security Advisory Database. https://rustsec.org/advisories.
[37] 2025. Stack canaries. https://en.wikipedia.org/wiki/Stack_buffer_overflow#Stack_canaries.
[38] 2025. Static and Dynamic Dispatch of Rust Trait Objects. https://doc.rust-lang.org/1.8.0/book/trait-objects.html.
[39] 2025. Terminator in rustc_middle::mir. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/terminator/struct.Terminator.html.
[40] 2025. TerminatorKind in rustc_middle::mir. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/mir/enum.TerminatorKind.html.
[41] 2025. To panic! or Not to panic! https://doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html.
[42] 2025. Toolchain Compatibility in Rust. https://doc.rust-lang.org/rustc/linker-plugin-lto.html#toolchain-compatibility.
[43] 2025. TyKind in rustc_middle::mir. https://doc.rust-lang.org/stable/nightly-rustc/rustc_middle/ty/sty/type.TyKind.html.
[44] 2025. Type checking in Rust. https://rustc-dev-guide.rust-lang.org/type-checking.html.
[45] 2025. Type propagation from MIR to LLVM IR in GSoC. https://discourse.llvm.org/t/gsoc2025-improve-rust-enzyme-reliability-and-compile-times/84523.
[46] 2025. The Unstable Features of Rust. https://doc.rust-lang.org/unstable-book/the-unstable-book.html.
[47] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.
[48] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.
[49] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[50] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2023. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–21.
[51] 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)*.
[52] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data flow sensitive fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.
[53] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[54] Shuang Hu, Baojian Hua, and Yang Wang. 2022. Comprehensiveness, Automation and Lifecycle: A New Perspective for Rust Security. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 982–991.
[55] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using input type inference to improve fuzzing. In *Proceedings of the 34th annual computer security applications conference*. 505–517.
[56] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia.
[57] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. Toronto, Canada.
[58] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided directed greybox fuzzing. In *Proceedings of the 30th USENIX Security Symposium (Security)*. Virtual Event.
[59] Wei Li, Dongjie He, Yujiang Gui, Wenguang Chen, and Jingling Xue. 2024. A context-sensitive pointer analysis framework for Rust and its application to call graph construction. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. 60–72.
[60] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 11th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Paderborn, Germany.
[61] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. Virtual Event, Korea.
[62] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. Pata: Fuzzing with path aware taint analysis. In *Proceedings of the 43nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[63] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Athens, Greece.
[64] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. Erasan: Efficient rust address sanitizer. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.
[65] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis. *Notes. Feb* (2012).
[66] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.
[67] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA, USA.
[68] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.
[69] 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 *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.
[70] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.
[71] Zhiwu Xu, Bohao Wu, Cheng Wen, Bin Zhang, Shengchao Qin, and Mengda He. 2024. RPG: Rust library fuzzing with pool-based fuzz target generation and generic support. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. Lisbon, Portugal.
[72] Xizhe Yin, Yang Feng, Qingkai Shi, Zixi Liu, Hongwang Liu, and Baowen Xu. 2024. FRIES: Fuzzing Rust Library Interactions via Efficient Ecosystem-Guided Target Generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1137–1148.

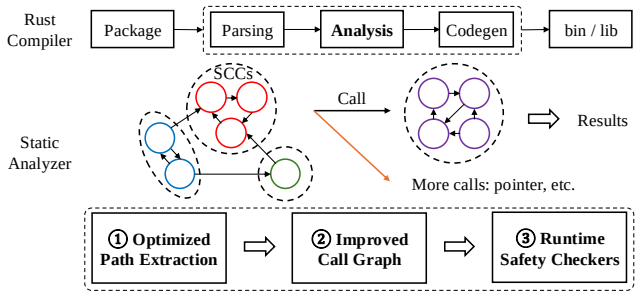


**Figure 3: The architecture of PanicChecker.**

# Appendices

## A Static Analysis

Identifying the runtime safety checks in the source code is challenging as they are inserted both explicitly and implicitly. We aim to perform static analysis in the Rust-specific IRs to precisely identify the Rust runtime safety checks, and study the prevalence and impacts of these checks. We develop PanicChecker, a path-sensitive and context-sensitive inter-procedural data-flow analysis tool on Rust MIR, to identify Rust runtime safety checks in programs. Figure 3 illustrates the framework of our approach. Given a bug location, PanicChecker analyzes all possible paths reaching this bug, and outputs the number of runtime safety checks inserted along these paths. Since PanicChecker is primarily used to motivate the research problem, we highlight three essential steps of our framework compared to existing state-of-the-art Rust static analyzers [50, 59, 61] as follows.

Our static analysis is based on Rust MIR, where runtime safety checks are inserted. It is mainly designed as follows. ① Optimized path extraction. PanicChecker adopts a path-sensitive approach. To mitigate the path explosion issue, we employ the Tarjan algorithm on the control-flow graphs (CFGs) to separate the strongly connected components (SCCs) [65] and unwind loops. This step enables the efficient enumeration of valuable paths. ② Improved call graph. We improve the inter-procedural call graph by considering Rust-specific features to achieve better precision. For both static and dynamic dispatch [38] function calls, we adopt an over-approximation strategy by adding calling relationships for all types that implement the corresponding traits. We also employ the design of Rupta [59] to conduct pointer analysis (*e.g.*, analyzing function pointers and pointer casts), thereby facilitating the identification of additional call edges. ③ Runtime safety check detection. Based on the constructed paths, PanicChecker iterates through each MIR instruction on the path, and determines whether it is an instruction of runtime safety check as shown in Table 2. It then outputs the number of runtime safety checks for each path, along with the average and total numbers. We implemented a prototype of our static analyzer PanicChecker with around 6,000 lines of Rust code. This helps precisely identify runtime safety checks in Rust MIR, and assess their impacts.