Understanding and Detecting Real-World Safety Issues in Rust

Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang

Abstract—Rust is a relatively new programming language designed for systems software development. Its objective is to combine the safety guarantees typically associated with high-level languages with the performance efficiency often found in executable programs implemented in low-level languages. The core design of Rust is a set of strict safety rules enforced through compile-time checks. However, to support more low-level controls, Rust also allows programmers to bypass its compiler checks by writing *unsafe* code. As the adoption of Rust grows in the development of safety-critical software, it becomes increasingly important to understand what safety issues may elude Rust's compiler checks and manifest in real Rust programs.

In this paper, we conduct a comprehensive, empirical study of Rust safety issues by close, manual inspection of 70 memory bugs, 100 concurrency bugs, and 110 programming errors leading to unexpected execution panics from five open-source Rust projects, five widely-used Rust libraries, and two online security databases. Our study answers three important questions: what memory-safety issues real Rust programs have, what concurrency bugs Rust programmers make, and how unexpected panics in Rust programs are caused. Our study reveals interesting real-world Rust program behaviors and highlights new issues made by Rust programmers. Building upon the findings of our study, we design and implement five static detectors. After being applied to the studied Rust programs and another 12 selected Rust projects, our checkers pinpoint 96 previously unknown bugs and report a negligible number of false positives, confirming their effectiveness and the value of our empirical study.

Index Terms—Rust, Memory Bug, Concurrency Bug, Bug Study, Static Bug Detection

1 Introduction

Rust [1] is a programming language specifically created for developing low-level software that is both efficient *and* safe [2], [3], [4], [5]. Its main idea is to inherit most features in C and C's good runtime performance but to mitigate C's safety concerns through strict compile-time checks. Over the past few years, Rust has experienced a significant surge in popularity [6], [7], [8], particularly in the realm of constructing low-level software such as operating systems and web browsers [9], [10], [11], [12], [13].

At the core of Rust's safety mechanisms lies the concept of *ownership*. The most basic ownership rule states that each value can have only one *owner* and the value is freed when its owner's *lifetime* ends. Rust builds upon this basic rule with a set of extended rules that still guarantee memory and thread safety. For example, the ownership can be *borrowed* or *moved*, multiple *aliases* can read a value at the same time, but at most one alias can write to a value at any time. These safety rules essentially prevent the combination of *aliasing* and *mutability*. They are checked at compile time to thwart severe memory and concurrency bugs from escaping into compiled programs. Rust also supports a panic mechanism, easily extendable by programmers to capture and prevent errors from causing further damage at runtime. By combining these elements, Rust achieves the runtime performance of its compiled programs as good as unsafe languages like C, but with much stronger safety guarantees.

- *B. Qin is with China Telecom Cloud Technology Co., Ltd, Beijing, China, 100083. E-mail: bobbqqin@gmail.com. This work was done during B. Qin's visit to the Pennsylvania State University as a Ph.D. student with Beijing University of Posts and Telecommunications.*
- *Y. Chen is with HoneycombData Inc, Santa Clara, CA, USA, 95054. E-mail: chen2709@purdue.edu. Y. Chen contributed equally with B. Qin in this work.*
- *H. Liu is with the University of Chicago, IL, USA, 60637. E-mail: haopeng.uc@gmail.com.*
- *H. Zhang and Q. Wen are with Beijing University of Posts and Telecommunications, Beijing, China, 100876. E-mail: {zhanghua288, wqy}@bupt.edu.cn.*
- *L. Song is with the Pennsylvania State University, State College, PA, USA, 16802. E-mail: songlh@ist.psu.edu.*
- *Y. Zhang is with the University of California San Diego, CA, USA, 92093. E-mail: yiying@ucsd.edu.*
- *Corresponding authors: Linhai Song and Yiying Zhang.*

While delivering safety, the safety rules enforced by Rust restrict programmers' control over low-level resources and frequently hinder their ability to implement desired functionalities. To grant programmers greater programming flexibility, Rust introduces the ability to bypass the main compiler safety checks by adding an *unsafe* label to Rust code. This label can be applied to either an entire function, by declaring it as unsafe, or to a specific piece of code within a function. In the latter case, the function can still be invoked as a safe function within safe code, which provides a way to encapsulate unsafe code. We refer to this code pattern as *interior unsafe*.

Unfortunately, the presence of unsafe code in Rust gives rise to safety concerns as it circumvents the safety checks performed by the Rust compiler. The inclusion of unsafe code and unsafe encapsulation complicates Rust's safety semantics. Furthermore, programming errors causing runtime panics can be exploited for denial-of-service (DoS) attacks, hurting Rust programs' reliability. Does unsafe code cause the same safety issues as traditional unsafe languages? Can there still be safety issues when programmers do not use any "unsafe" label in their code? What happens when unsafe and safe code interact? What are the common programming errors leading to runtime panics? Several recent works [14],

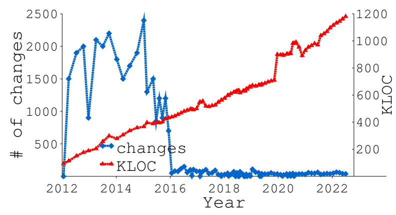


Fig. 1: Rust History. *(Each blue point shows the number of feature changes in one release version. Each red point shows total LOC in one release version.)*

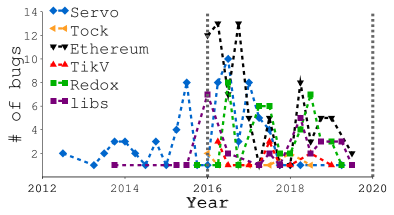


Fig. 2: Time of Studied Bugs. *(Each point shows the number of our studied bugs that were patched during a time period of three months.)*

TABLE 1: Studied Applications and Libraries. *(The start time, the number of stars, and commits on GitHub, total lines of source code, the number of memory safety bugs, blocking bugs, non-blocking bugs, and panic bugs. libraries: maximum values among our studied libraries. There are 27 bugs collected from the two CVE databases.)*

Software	Start Time	Stars	Commits	LOC	Mem	Blk	NBlk	Panic
Servo	2012/02	14574	38096	271K	14	13	18	29
Tock	2015/05	1343	4621	60K	5	0	2	2
Ethereum	2015/11	5565	12121	145K	2	34	4	41
TiKV	2016/01	5717	3897	149K	1	4	3	6
Redox	2016/08	11450	2129	199K	20	2	3	20
libraries	2010/07	3106	2402	25K	7	6	10	7

[15], [16], [17] formalize and theoretically prove (a subset of) Rust's safety and interior-unsafe mechanisms. However, it remains unclear how Rust's language safety and unsafe designs impact real-world Rust developers and what safety issues occur in real Rust software. With the wider adoption of Rust in systems software in recent years, it is crucial to answer these questions.

In this paper, we undertake an empirical investigation into safety issues in real-world Rust programs. We analyze the interplay between safe and unsafe code, exploring how their interaction can lead to memory safety issues (*i.e.*, illegal memory accesses), thread safety issues (*i.e.*, thread synchronization issues like deadlocks and race conditions), and unexpected program panics. Our study places specific emphasis on the influence of Rust's ownership and lifetime rules, as well as Rust's other distinctive and crucial language features, on developers' programming, and how the misuse of these features causes safety issues.

Our study covers five Rust-based systems and applications (two OSes, a browser, a key-value store system, and a blockchain system), five widely-used Rust libraries, and two online vulnerability databases. We scrutinize their source code, their GitHub commit logs and publicly reported bugs by first filtering them into a small relevant set and then manually inspecting this set. Overall, we investigate a total of 70 memory-safety issues, 100 thread-safety issues, and 110 panic issues, as shown in Table 1.

Our study consists of three parts, and we have made contribution in each of them. First, we study memory-safety issues in real Rust programs by inspecting bugs in our selected applications and libraries and by examining *all* Rust issues reported on CVE [18] and RustSec [19] prior to Jun 2019. Our analysis not only focuses on understanding these bugs' behaviors but also delves into how their root causes are propagated to their effects. We find that all memory-safety bugs involve unsafe code, and (surprisingly) most of them also involve safe code. Mistakes tend to occur when programmers write safe code without taking sufficient caution regarding related unsafe code, whether within the same function or in improperly encapsulated interior-unsafe functions. Additionally, we observe that the scope of *lifetime* in Rust is challenging to reason about, particularly when combined with unsafe code. Incorrect understanding of *lifetime* causes many memory-safety issues.

Second, we investigate concurrency bugs, including both blocking bugs where one or more threads unintentionally become stuck, and non-blocking bugs where multiple threads complete their execution but yield undesired results [20]. Strikingly, our research reveals that non-blocking bugs can happen in both unsafe and safe code and that *all* blocking bugs we have studied are in safe code. Although many bug patterns in Rust follow conventional concurrency bug patterns (*e.g.*, double lock, atomicity violation), a significant portion of concurrency bugs in Rust arise from programmers' misunderstanding of Rust's (complex) lifetime and safety rules.

Finally, we study programming errors causing unexpected execution panics. We find that (almost) all the errors happen in safe code. A large portion of them can be attributed to mistakes made when using Rust's libraries (`Result` and `Option`) to handle runtime errors. Moreover, the Rust runtime detects overflow when performing various arithmetic operations and trigger panics accordingly. This functionality behaves differently on different numeric types and different program building modes. Without enough caution, it is quite easy to misjudge the runtime's behavior, causing unexpected panics.

Regarding all the three aspects mentioned above, we provide insightful suggestions to future Rust programmers and language designers. Most of these suggestions can be directly acted on. To illustrate, based on our summary of common buggy code patterns and pitfalls, we offer recommendations on good programming practices and concrete suggestions on the design of future Rust bug detectors[1] and programming tools. We firmly believe that programmers, researchers, and language designers can utilize our study results and the concrete, actionable suggestions we propose to improve Rust software development, including but not limited to better programming practices, better bug detection tools, and better language designs [21], [22], [23], [24], [25], [26], [27], [28].

Using the results from our empirical study, we further conduct Rust bug detection by developing five static detectors. These detectors specifically target use-after-free bugs, double-lock bugs, conflicting-ordering deadlocks, atomicity violations, and code sites that can trigger runtime panics. In contrast to linters that analyze program source code [29], our detectors conduct ownership- and lifetime-related analyses on Rust's mid-level intermediate representation (MIR). They pinpoint code snippets exhibiting the buggy patterns identified in our empirical study. Programmers can enhance the safety of their programs by addressing the reported memory

1. We use detectors and checkers exchangeably in this paper.

bugs or concurrency bugs. Additionally, they can inspect the context of the identified panic sites and modify their programs to create panic-free versions for improved reliability. In addition to all our studied Rust applications and libraries, we also apply our detectors to 12 additionally selected Rust programs. In total, these detectors discover 96 previously unknown bugs in real Rust applications, with only seven false positives reported. We have notified the programmers about all the detected bugs. So far, programmers have fixed 45 bugs based on our reporting and confirmed another 41 bugs as real bugs. Comparing the current state-of-the-art Rust bug detection technique [30], our detectors outperform in terms of detecting a higher number of bugs, while producing fewer false positives. The effectiveness and accuracy of our detectors validate the value of our empirical study.

Overall, this paper makes the following contributions.

- A comprehensive study on 70 real Rust memory-safety issues, 100 concurrency bugs, and 110 panic issues.
- Ten insights and five suggestions that can help Rust programmers and the future development of Rust.
- Five novel Rust bug detectors and 96 detected bugs in popular Rust programs.

This article builds upon and enhances a previously published conference paper [31]. Specifically, we study 110 panic issues in real-world Rust programs. We inspect the issues' root causes and fix strategies. We redesign the two bug detectors in the previous paper. Notably, we have successfully eliminated the false positives reported by the use-after-free detector, and significantly enhanced the effectiveness of the double-lock detector (as shown by the number of detected bugs in the same set of applications). Additionally, we design and implement three new detectors to cover two types of concurrency bugs and panic sites. Furthermore, we perform thorough experiments to evaluate the detectors by running them on both the studied applications and 12 additionally selected programs, and comparing our detectors with the state-of-the-art Rust bug detection technique called Rudra [30]. Finally, we make a large number of changes to improve the article's presentation and discuss recent research advancements in the related areas.

We have released our study results, source code of our bug detectors, and detailed experimental results, all of which can be found at https://github.com/BurtonQin/lockbud.

2 Background and Related Work

This section provides some background of Rust, encompassing its history, safety (and unsafe) mechanisms, supported runtime errors, and associated error handling methods.

2.1 Language Overview and History

Rust is a type-safe language that prioritizes both efficiency and safety. It was designed for low-level software development where programmers seek low-level control of resources (so that programs run efficiently) but want to be type-safe and memory-safe. Rust defines a set of strict safety rules and uses the compiler to check these rules to statically detect and prevent numerous potential safety issues. At runtime, Rust behaves similarly to C and can achieve a performance level that is comparable to it.

```
1  #[derive(Debug)]
2  struct Test {v: Vec<i32>}
3  fn f0(_t: Test) {}
4  fn f1() {
5   let t0 = Test{v: vec![0]};
6   f0(t0);
7   // println!("{:?}", t0);
8   if true {
9    let t1 = Test{v: vec![1]};
10  }
11  // println!("{:?}", t1);
12 }
```

(a) ownership & lifetime

```
13 fn f2() {
14  let mut t2 = Test{v: 2};
15  let r1 = &t2;
16  let mut r2 = &mut t2;
17  r2.v = 3;
18  // println!("{:?}", r1);
19 }
```

(b) borrow

Fig. 3: Sample code to illustrate Rust's safety rules.

Rust has consistently been ranked as the most beloved programming language since 2016, based on Stack Overflow surveys [6], [7], [8], [32], [33], [34]. Additionally, it gained recognition as the fifth fastest-growing language on GitHub in 2018 [35]. Because of its safety and performance benefits, Rust's adoption in systems software has increased rapidly in recent years [9], [12], [13], [36], [37], [38], [39]. Prominent examples include Microsoft actively exploring Rust as an alternative to C/C++ because of its memory-safety features [40], [41], Amazon extensively utilizing Rust in AWS for implementing performance-sensitive components [42], and Google implementing low-level Android components in Rust [43].

Rust was first released in 2012 and has now reached version 1.68.0. Figure 1 shows the progression of Rust's feature changes and lines of code (LOC) over its history. Rust went through heavy changes in the first four years since its release, and it has been stable since Jan 2016 (v1.6.0). Having maintained stability for over six years, we consider Rust to be sufficiently mature for empirical studies such as ours. Figure 2 shows when the analyzed bugs were resolved. Out of the 280 bugs, 244 of them were fixed after 2016. Therefore, we believe our study results reflect the issues observed in stable Rust versions.

2.2 Safety Mechanisms

The goal of Rust's *safety* mechanism is to prevent memory and thread safety issues that have plagued C programs. Its design centers around the notion of *ownership*. At its core, Rust enforces a strict and restrictive rule of ownership: each value has one and only one *owner* variable, and when the owner's *lifetime* ends, the value will be *dropped* (freed). The lifetime of a variable is the scope where it is valid, *i.e.*, from its creation to the end of the function it is in or to the end of matching parentheses (*e.g.*, the lifetime of `t1` in Figure 3 spans from line 9 to line 10). This strict ownership rule eliminates memory errors like use-after-free and double-free, since the Rust compiler can statically detect and reject the use of a value when its owner goes out of scope (*e.g.*, uncommenting line 11 in Figure 3 will raise a compile error as in C/C++). This rule also eliminates synchronization errors like race conditions, since only one thread can own a value at a time.

Under Rust's basic ownership rule, a value has one exclusive owner. Rust extends this basic rule with a set of features to support more programming flexibility while still ensuring memory- and thread-safety. These features (as explained below) relax the restriction of having only one

```
1 struct TestCell { value: i32, }
2 unsafe impl Sync for TestCell{}
3 impl TestCell {
4   fn set(&self, i: i32) {
5     let p = &self.value as * const i32 as * mut i32;
6     unsafe{*p = i};
7   }
8 }
```

Fig. 4: A bad practice for using (interior) unsafe.

owner for the lifetime of a value but still *prohibit having aliasing and mutation at the same time*, and Rust statically checks these extended rules at compile time.

Ownership move. The ownership of a value can be *moved* from one *scope* to another (*e.g.*, from a caller to a callee, from one thread to another thread) or from one owner variable to a different one. The Rust compiler statically guarantees that an owner variable cannot be accessed after its ownership is moved. As a result, a caller cannot access a value anymore if the value is dropped in the callee function, and a shared value can only be owned by one thread at any time. For example, if line 7 in Figure 3 is uncommented, the Rust compiler will report an error, since the ownership of `t0` has already been moved to function `f0()` at line 6. The compiler prevents a use-after-free, since `t0` and the heap buffer associated with its `Vec` field are dropped inside `f0()`. Attempting to print `t0` at line 11 would also access the dropped heap buffer [2].

Ownership borrowing. A value's ownership can also be *borrowed* temporarily to another variable for the lifetime of this variable without moving the ownership. Borrowing is achieved by passing the value by reference to the borrower variable. Without using particular libraries, Rust does not allow borrowing ownership across threads, since a value's lifetime cannot be statically inferred across threads and there is no way the Rust compiler can guarantee that all usages of a value are covered by its lifetime.

Mutable and Shared references. Rust offers two types of references. It allows multiple shared read-only references, *i.e.*, immutable references permitting read-only aliasing. Additionally, a value's reference can also be *mutable*, enabling write access to the value. However, there can only be one mutable reference and no immutable references simultaneously. Once a value's ownership is borrowed through a *mutable reference*, the temporary owner gains exclusive write access to the value. In Figure 3, an immutable reference (`r1`) and a mutable reference (`r2`) are created at line 15 and line 16, respectively. The Rust compiler does not allow line 18, since it will make the lifetime of `r1` end after line 18, making `r1` and `r2` co-exist at line 16 and line 17.

2.3 Unsafe and Interior Unsafe

Rust's safety rules are strict and its static compiler checking for the rules is conservative. Developers (especially low-level software developers) often need more flexibility in writing their code, and some desire to manage safety by themselves. Rust allows programs to bypass its safety checking with the *unsafe* feature, denoted by the label `unsafe`. A function can be marked as `unsafe`; a piece of code can be marked as `unsafe`; and a *trait* can be marked as `unsafe`[3].

Code regions marked with `unsafe` will bypass Rust's compiler checks and be able to perform five types of functionalities: dereferencing and manipulating raw pointers, accessing and modifying mutable static variables (*i.e.*, global variables), calling unsafe functions, implementing unsafe traits, and accessing `union` fields. As shown by Figure 4, `TestCell` implements the unsafe `Sync` trait at line 2, so that a `TestCell` object can be shared between two threads within safe code after being declared with `Arc`. The pointer operation at line 6 is in an unsafe code region.

Rust allows a function to have unsafe code only internally; such a function can be called by safe code and thus is considered "safe" externally. We call this pattern *interior unsafe* (*e.g.*, function `set()` in Figure 4). Many APIs provided by the Rust standard library are interior-unsafe functions, such as `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, and `RwLock`.

The design rationale of interior unsafe code is to have the flexibility and low-level management of unsafe code but to encapsulate the unsafe code in a carefully-controlled interface, or at least that is the intention of the interior-unsafe design. For example, Rust uses interior-unsafe functions to allow the combination of aliasing and mutation (*i.e.*, bypassing Rust's core safety rules) in a controlled way: the internal unsafe code can mutate values using multiple aliases, but these mutations are encapsulated in a small number of immutable APIs that can be called in safe code and pass Rust's safety checks. Rust calls this feature *interior mutability*.

Programmers must exercise sufficient caution when using interior mutability functions in multi-threaded contexts to avoid introducing concurrency bugs. Figure 4 illustrates a bad practice. In the example, the `set()` function borrows its input `self` immutably, but it modifies the `value` field of `self` through the pointer `p` (an alias) at line 6. Since `TestCell` implements the `Sync` trait, a `TestCell` object can be shared across threads after being declared with `Arc`. Additionally, as `set()` borrows `self` immutably, the Rust compiler does not prevent concurrent calls to `set()`, potentially leading to data races at line 6. As we will observe later, many concurrency bugs stem from a similar cause.

2.4 Error and Error Handling

Rust distinguishes between two types of runtime errors: unrecoverable errors and recoverable errors. An unrecoverable error results in the termination of program execution, while a recoverable error allows the program to take suitable measures to handle the error and proceed with its execution.

Rust programmers can use the `panic!` macro to trigger unrecoverable errors[4] for cases that cannot be reasonably handled. When invoked, this macro generates an error message, cleans up the stack (*e.g.*, dropping live objects on the stack), and halts the program's execution. The Rust standard library commonly invokes `panic!`, when library users conduct dangerous or insecure actions, such as accessing an array with an out-of-bounds index. Additionally, the Rust compiler automatically injects sanitizing code for certain mathematical

2. The heap memory associated with the `Vec` field of `t0` is not copied during the execution, since struct `Test` does not derive the `Copy` trait.

3. Rust traits are similar to interfaces in traditional languages like Java.

4. We will use unrecoverable errors and panics interchangeably in this paper.

```
1  fn take_input() -> Result<char, Error> {
2    let input = user_input();
3    match input {
4      'a'..='z' => Ok(input),
5      'A'..='Z' => Err(CapitalLetterError),
6      _ => Err(InvalidLetterError),
7    }
8  }
9
10 fn main() {
11   match take_input() {
12     Ok(input) => println!("Valid␣Letter:␣{}", input),
13     Err(CapitalLetterError) => println!("Capital␣Letter!"),
14     Err(InvalidLetterError) => println!("Invalid␣Letter!"),
15   };
16 }
```

Fig. 5: Sample code for a recoverable error.

operations (especially in the debug build mode), which triggers panics upon detecting arithmetic errors such as division by zero or integer overflow.

Rust offers two `enum` types, `Result` and `Option`, for callee functions to communicate recoverable errors to their callers. When a function returns a `Result` object, it can have one of two possible values. The first value is `Ok(T)`, where `T` is a generic type and is instantiated as the return type required by the function's functionality, indicating that the function executes successfully. On the other hand, the `Result` can also be `Err(E)`, denoting the function encounters an error and requires its caller to handle the error, since the caller may have more context information. Similarly, an `Option` object can be `Some(T)` or `None` for a successful or an unsuccessful function execution, respectively. Rust programmers can use the `match` or `if` statements to check the value of a `Result` or an `Option` object and take proper actions accordingly. For example, in Figure 5, function `take_input()` takes a character from the program user at line 2. The function only expects a lowercase letter, and thus, if the character falls within the range of 'a' to 'z', the function returns `Ok(input)` at line 4. In case the user inputs a capital letter, the function returns `Err(CapitalLetterError)` at line 5. For all other cases, it returns `Err(InvalidLetterError)` at line 6. The calling function (function `main()`) leverages a `match` statement (lines 11–15) to process the return of `take_input()` and outputs corresponding messages.

Rust offers convenient shortcuts for frequently used programming patterns when working with `Result` or `Option` objects. For instance, if a programmer is certain that a `Result` must be `Ok(T)`, the programmer can call `Result.unwrap()` to directly access the value inside `Ok(T)`. However, it is important to note that `Result.unwrap()` panics, if the programmer's assumption is incorrect and the `Result` is actually `Err(E)`. In Section 6, we will see that a lot of unexpected panics occur due to unwrapping `Result` or `Option` objects directly when programmers mistakenly assume they hold valid values.

3 Study Methodology

Although there are books, blogs, and theoretical publications that discuss Rust's design philosophy, benefits, and unique features, it is unclear how real-world Rust programmers use Rust and what pitfalls they make. An empirical study on real-world Rust bugs like ours can also reveal what mistakes (bugs) real programmers make and how they fix them. Some of these mistakes could be previously unknown. Even if they are, we can demonstrate through real data how often they happen and dig into deeper reasons why programmers make mistakes in that way. Our study reflects all the above values of empirical studies.

To perform an empirical study, we spent numerous manual efforts inspecting and understanding real Rust code. These efforts result in this paper, which we hope will fuel future research and practices to improve Rust programming and in turn save future Rust programmers' time. Before presenting our study results, this section first outlines our studied applications and our study methodology.

Studied Rust software and libraries. Our criteria of selecting what Rust software to study include open source, long code history, popular software, and active maintenance. We also aim to cover a wide range of software types (from user-level applications and libraries to OSes). Based on these criteria, we selected five software systems and five libraries for our study (Table 1).

Servo [9] is a browser engine developed by Mozilla. Servo has been developed side by side with Rust and has the longest history among the applications we studied. *TiKV* [39] is a key-value store that supports both single key-value-pair and transactional key-value accesses. *Parity Ethereum* [44] is a fast, secure blockchain client written in Rust (we call it *Ethereum* for brevity in the rest of the paper). *Redox* [13] is an open-source secure OS that adopts microkernel architecture but exposes UNIX-like interface. *Tock* [45] is a Rust-based embedded OS. Tock leverages Rust's compile-time memory-safety checking to isolate its OS modules.

Apart from the above five applications, we studied five widely-used Rust libraries. They include 1) *Rand* [46], a library for random number generation, 2) *Crossbeam* [47], a framework for building lock-free concurrent data structures, 3) *Threadpool* [48], Rust's implementation of thread pool, 4) *Rayon* [49], a library for parallel computing, and 5) *Lazy_static* [50], a library for defining lazily evaluated static variables.

Collecting and studying bugs. To collect bugs, we analyzed GitHub commit logs from the applications in Table 1. We first filtered the commit logs using a set of bug-related keywords, *e.g.*, "use-after-free" for memory bugs, "deadlock" for concurrency bugs, "overflow" for panics. These keywords either cover important issues in the research community [51], [52], [53] or are used in previous works to collect bugs [20], [54], [55], [56]. We then manually inspected filtered logs to identify bugs. We also analyzed all Rust-related vulnerabilities prior to Jun 2019 in two online vulnerability databases, CVE [18] and RustSec [19].

The scope of panic issues overlaps with that of memory bugs and concurrency bugs. For instance, a runtime panic can be triggered by an out-of-bounds memory access through a `Vec![T]`, and a panic may result from two threads holding mutable references to the same `RefCell` at the same time. To distinguish a panic issue from a memory bug, we assess whether an invalid memory access occurs earlier or if a panic occurs earlier. If an invalid memory access occurs first, we categorize the issue as a memory bug; otherwise, we classify it as a panic issue. To separate a panic issue

TABLE 2: Memory Bugs Category. *(Buffer: Buffer overflow; Null: Null pointer dereferencing; Uninitialized: Reading uninitialized memory; Invalid: Invalid free; UAF: Use after free. ⋆: numbers in () are for bugs whose effects are in interior-unsafe functions.)*

Category	Wrong Access			Lifetime Violation			Total
	Buffer	Null	Uninitialized	Invalid	UAF	Double free	
safe	0	0	0	0	1	0	1
unsafe ⋆	4 (1)	12 (4)	0	5 (3)	2 (2)	0	23 (10)
safe → unsafe ⋆	17 (10)	0	0	1	11 (4)	2 (2)	31 (16)
unsafe → safe	0	0	7	4	0	4	15

```
1 pub struct FILE {
2     buf: Vec<u8>,
3 }
4
5 pub unsafe fn _fdopen(...) {
6     let f = alloc(size_of::<FILE>()) as * mut FILE;
7 -   *f = FILE{buf: vec![0u8; 100]};
8 +   ptr::write(f, FILE{buf: vec![0u8; 100]});
9 }
```

Fig. 6: An invalid-free bug in Redox.

from a concurrency bug, we primarily examine whether a bug is confined to a single thread. In total, we examined 70 memory bugs, 100 concurrency bugs, and 110 unexpected panic issues.

We manually inspected and analyzed all available sources for each bug, including its patch, bug report, and online discussions. Each bug is examined by at least two people in our team. We also reproduced a set of bugs to validate our understanding.

Instead of selecting some of the study results (*e.g.*, those that are unexpected), we report *all* our study results and findings. Doing so can truthfully reflect the actual status of how programmers in the real world use Rust. During our bug study, we identified common mistakes made by different developers in different projects. We believe similar mistakes can be made by other developers in many other Rust projects. Reporting all found errors (including known errors) can help developers avoid similar errors in the future and motivate the development of related detection techniques.

4 Memory Safety Issues

Memory safety is a key design goal of Rust. Rust uses a combination of static compiler checks and dynamic runtime checks to ensure memory safety for its safe code. However, it is not clear whether or not there are still memory-safety issues in real Rust programs, especially when they commonly include unsafe and interior-unsafe code. This section presents our detailed analysis of 70 real-world Rust memory-safety issues and their fixes.

4.1 Bug Analysis Results

It is important to understand both the *cause* and the *effect* of memory-safety issues (bugs). We categorize our studied bugs along two dimensions: how errors propagate and what are the effects of the bugs. Table 2 summarizes the results in the two dimensions introduced above.

For the first dimension, we analyze the error propagation chain from a bug's cause to its effect and consider how safety semantics change during the propagation chain. Similar to prior bug analysis methodologies [57], [58], we consider the code where a bug's patch is applied as its cause and the code where the error symptom can be observed as its effect. Based on whether cause and effect are in safe or unsafe code, we categorize bugs into four groups: safe → safe (or simply, safe), safe → unsafe, unsafe → safe, and unsafe → unsafe (or simply, unsafe).

For the second dimension, we categorize bug effects into wrong memory accesses (*e.g.*, buffer overflow) and lifetime violations (*e.g.*, use after free).

Buffer overflow. 17 out of 21 bugs in this category follow the same pattern: an error happens when computing buffer size or index in safe code and an out-of-boundary memory access happens later in unsafe code. For 11 bugs, the effect is inside an interior unsafe function. Six interior unsafe functions contain condition checks to avoid buffer overflow. However, the checks do not work due to wrong checking logic, inconsistent struct status, or integer overflow. For three interior unsafe functions, their input parameters are used directly or indirectly as an index to access a buffer, without any boundary checks.

Null pointer dereferencing. Rust allows the assignment of a pointer value to a raw pointer or comparison of a raw pointer with a value in safe code, but it does not permit dereferencing a raw pointer in safe code. Thus, all bugs in this category are caused by dereferencing a null pointer in unsafe code. In four of them, null pointer dereferencing happens in an interior unsafe function. These interior unsafe functions do not perform proper checking before the dereferencing.

In addition to raw pointers, Rust also supports references and standard library `Box`, each type with usage scenarios similar to pointers. However, the Rust compiler does not permit variables of these types to be NULL, eliminating the possibility of null pointer dereferencing on these types. In situations where a Box variable might be invalid, programmers can employ an `Option` object, with the `Box` variable serving as the internal value of `Some(T)` and `None` indicating that the `Box` variable is invalid.

Reading uninitialized memory. All the seven bugs in this category are unsafe → safe. Four of them use unsafe code to create an uninitialized buffer and later read it using safe code. The rest initialize buffers incorrectly, *e.g.*, using memcpy with wrong input parameters.

Invalid free. Out of the ten invalid-free bugs, five share the same (unsafe) code pattern. Figure 6 shows one such example. The variable `f` is a pointer pointing to an uninitialized memory buffer with the same size as struct `FILE` (line 6). Assigning a new `FILE` struct to `*f` at line 7 ends the lifetime of the previous struct `f` points to, causing the previous struct to be dropped by Rust. All the allocated memory with the previous struct will be freed, (*e.g.*, memory in `buf` at line 2). However, since the previous struct contains an uninitialized memory buffer, freeing its heap memory is invalid. Note that such behavior is unique to Rust and does not happen in traditional languages (*e.g.*, `*f=buf` in C/C++ does not cause the object pointed by `f` to be freed).

Use after free. 11 out of 14 use-after-free bugs happen because an object is dropped implicitly in safe code (when its lifetime ends), but a pointer to the object or to a field of the object still exists and is later dereferenced in unsafe code. Figure 7 shows

```
1   pub fn sign(data: Option<&[u8]>) {
2 -   let p = match data {
3 -     Some(data) => BioSlice::new(data).as_ptr(),
4 -     None => ptr::null_mut(),
5 -   };
6 +   let bio = match data {
7 +     Some(data) => Some(BioSlice::new(data)),
8 +     None => None,
9 +   };
10 +  let p = bio.map_or(ptr::null_mut(),|p| p.as_ptr());
11    unsafe {
12      let cms = cvt_p(CMS_sign(p));
13    }
14  }
```

Fig. 7: A use-after-free bug in RustSec.

an example. When the input `data` is valid, a `BioSlice` object is created at line 3 and its address is assigned to a pointer `p` at line 2. `p` is used to call an unsafe function `CMS_sign()` at line 12 and it is dereferenced inside that function. However, the lifetime of the `BioSlice` object ends at line 5, where its implicit owner variable's valid scope concludes, leading to the object being dropped there. The use of `p` is thus after the object has been freed. The patch resolves this issues by moving the `BioSlice` object to `bio` at line 6, and thus extends the object's lifetime to the end of the function, encompassing the call of function `CMS_sign()` at line 12. Both this bug and the bug in Figure 6 are caused by wrong understanding of object lifetime. We have identified misunderstanding of lifetime being the main reason for most use-after-free and many other types of memory-safety bugs.

There is one use-after-free bug whose cause and effect are both in safe code. This bug occurred with an early Rust version (v0.3) and the buggy code pattern is not allowed by the Rust compiler now. The last two bugs happen in a self-implemented vector. Developers explicitly drop the underlying memory space in unsafe code due to some error in condition checking. Later accesses to the vector elements in (interior) unsafe code trigger a use-after-free error.

Double free. There are six double-free bugs. Other than two bugs that are safe → unsafe and similar to traditional double-free bugs, the rest are all unsafe → safe and unique to Rust. These buggy programs first conduct some unsafe memory operations to create two owners of a value. When these owners' lifetime ends, their values will be dropped (twice), causing double free. One such bug is caused by

```
t2 = ptr::read::<T>(&t1)
```

which is a `unsafe` function and reads the content of `t1` and puts it into `t2` without moving `t1`. If type `T` contains a pointer field that points to some object, the object will have two owners, `t1` and `t2`. When `t1` and `t2` are dropped by Rust implicitly when their lifetime ends, double free of the object happens. A safer way is to move the ownership from `t1` to `t2` using `t2 = t1`. These ownership rules are unique to Rust and programmers need to be careful when writing similar code.

Insight 1: *Rust's safety mechanisms (in Rust's stable versions) are very effective in preventing memory bugs. All memory-safety issues involve unsafe code (although many of them also involve safe code).*

Suggestion 1: *Future memory bug detectors can ignore safe code that is unrelated to unsafe code to reduce false positives and to improve execution efficiency.*

Interior unsafe functions conceal their internal unsafe code beneath safe APIs. Consequently, meticulously crafted interior unsafe functions should be regarded as safe functions. However, self-implemented interior unsafe functions demand careful scrutiny and testing. For example, out of the 26 memory bugs whose effects manifest in an interior-unsafe function, the interior unsafe function is implemented by the project programmers themselves for 25 bugs, with the remaining one originating from the standard library [59].

4.2 Fixing Strategies

We categorize the fixing strategies of our collected memory-safety bugs into four categories.

Conditionally skip code. 30 bugs were fixed by capturing the conditions that lead to dangerous operations and skipping the dangerous operations under these conditions. For example, when the offset into a buffer is outside its boundary, buffer accesses are skipped. 25 of these bugs were fixed by skipping unsafe code, four were fixed by skipping interior unsafe code, and one skipped safe code.

Adjust lifetime. 22 bugs were fixed by changing the lifetime of an object to avoid it being dropped improperly. These include extending the object's lifetime to fix use-after-free (*e.g.*, the fix of Figure 7), changing the object's lifetime to be bounded to a single owner to fix double-free, and avoiding the lifetime termination of an object when it contains uninitialized memory to fix invalid free (*e.g.*, the fix of Figure 6).

Change unsafe operands. Nine bugs were fixed by modifying operands of unsafe operations, such as providing the right input when using memcpy to initialize a buffer and changing the length and capacity into a correct order when calling `Vec::from_raw_parts()`.

Other. The remaining nine bugs used various fixing strategies outside the above three categories. For example, one bug was fixed by correctly zero-filling a created buffer. Another bug was fixed by changing memory layout.

Insight 2: *More than half of memory-safety bugs were fixed by changing or conditionally skipping unsafe code, but only a few were fixed by completely removing unsafe code, suggesting that unsafe code is unavoidable in many cases.*

Based on this insight, we believe that it is promising to apply existing techniques [60], [61] that synthesize conditions for dangerous operations to fix Rust memory bugs.

5 Thread Safety Issues

Rust provides unique thread-safety mechanisms to help prevent concurrency bugs, and as Rust language designers put it, to achieve "fearless concurrency" [62]. However, we have found a fair amount of concurrency bugs in Rust programs. Similar to a recent work's taxonomy of concurrency bugs [20], we divide our 100 collected concurrency bugs into blocking bugs (*e.g.*, deadlocks) and non-blocking bugs (*e.g.*, data races).

This section presents our analysis on the root causes and fixing strategies of our collected blocking and non-blocking bugs, with a particular emphasis on how Rust's ownership and lifetime mechanisms and its unsafe usages impact concurrent programming.

TABLE 3: Types of Synchronization in Blocking Bugs.

Software	Mutex&Rwlock	Condvar	Channel	Once	Other
Servo	6	0	5	0	2
Tock	0	0	0	0	0
Ethereum	27	6	0	0	1
TiKV	3	1	0	0	0
Redox	2	0	0	0	0
libraries	0	3	1	1	1
Total	38	10	6	1	4

5.1 Blocking Bugs

Blocking bugs manifest when one or more threads conduct operations that wait for resources (blocking operations), but these resources are never available. In total, we studied 59 blocking bugs. All of them are caused by using interior unsafe functions in safe code. Among them, 25 are due to misuse of the standard library, and the remaining 34 are due to misuse of the third-party library.

Bug Analysis. We study blocking bugs by examining what blocking operations programmers use in their buggy code and how the blocking conditions happen. Table 3 summarizes the number of blocking bugs that are caused by different blocking operations. 55 out of 59 blocking bugs are caused by operations of synchronization primitives, like `Mutex` and `Condvar`. All these synchronization operations have safe APIs, but their implementation heavily uses interior-unsafe code, since they are primarily implemented by reusing existing libraries like pthread. The other four bugs are not caused by primitives' operations (one blocked at an API call only on Windows platform, two blocked at a busy loop, and one blocked at `join()` of threads).

`Mutex` and `RwLock`. Different from traditional multi-threaded programming languages, the locking mechanism in Rust is designed to protect data accesses, instead of code fragments [63]. To allow multiple threads to have write accesses to a shared variable in a safe way, Rust developers can declare the variable with both `Arc` and `Mutex`. The `lock()` function returns a `LockGuard` object with a reference to the shared variable and *locks* it. The Rust compiler verifies that all accesses to the shared variable are conducted with the lock being held, guaranteeing mutual exclusion. A lock is automatically released when the lifetime of the returned `LockGuard` ends (the Rust compiler implicitly calls `unlock()` when the lifetime ends).

Failing to acquire `lock` (for `Mutex`) or `read/write` (for `RwLock`) results in thread blocking for 38 bugs, with 30 of them caused by double locking, seven caused by acquiring locks in conflicting orders, and one caused by forgetting to unlock when using a self-implemented mutex. Even though problems like double locking and conflicting lock orders are common in traditional languages too, Rust's complex lifetime rules together with its implicit unlock mechanism make it harder for programmers to write blocking-bug-free code.

Figure 8 shows a double-lock bug. The variable `client` is an `Inner` object protected by an `RwLock`. At line 3, its `read` lock is acquired and its `m` field is used as input to call function `connect()`. If `connect()` returns `Ok`, the `write` lock is acquired at line 7 and the `inner` object is modified at line 8. The `write` lock at line 7 will cause a double lock, since the lifetime of the temporary lock guard object returned by `client.read()` spans the whole `match` code block and

```
1    fn do_request() {
2      //client: Arc<RwLock<Inner>>
3  -   match connect(client.read().unwrap().m) {
4  +   let result = connect(client.read().unwrap().m);
5  +   match result {
6        Ok(_) => {
7          let mut inner = client.write().unwrap();
8          inner.m = mbrs;
9        }
10       Err(_) => {}
11     };
12   }
```

Fig. 8: A double-lock bug in TiKV.

the read lock is held until line 11. The patch is to save to the return of `connect()` to a local variable to release the read lock at line 4, instead of using the return directly as the condition of the `match` code block.

This bug demonstrates the unique difficulty in knowing the boundaries of critical sections in Rust. Rust developers need to have a good understanding of the lifetime of a lock guard variable returned by `lock()`, `read()`, or `write()` to know when `unlock()` will implicitly be called. But Rust's complex language features make it tricky to determine lifetime scope. For example, in six double-lock bugs, the first lock is in a `match` condition and the second lock is in the corresponding `match` body (*e.g.*, Figure 8). In another five double-lock bugs, the first lock is in an `if` condition, and the second lock is in the `if` block or the `else` block. The unique nature of Rust's locking mechanism to protect data accesses makes the double-lock problem even more severe, since mutex-protected data can only be accessed after calling `lock()`.

`Condvar`. In eight of the ten bugs related to `Condvar`, one thread is blocked at `wait()` of a `Condvar`, while no other threads invoke `notify_one()` or `notify_all()` of the same `Condvar`. In the other two bugs, one thread is waiting for a second thread to release a lock, while the second thread is waiting for the first to invoke `notify_all()`.

Channel. In Rust, a channel has unlimited buffer size by default, and pulling data from an empty channel blocks a thread until another thread sends data to the channel. There are five bugs caused by blocking at receiving operations. In one bug, one thread blocks at pulling data from a channel, while no other threads can send data to the channel. For another three bugs, two or more threads wait for data from a channel but fail to send data other threads wait for. In the last bug, one thread holds a lock while waiting for data from a channel, while another thread blocks at lock acquisition and cannot send its data.

Rust also supports channel with a bounded buffer size. When the buffer of a channel is full, sending data to the channel will block a thread. There is one bug that is caused by a thread being blocked when sending to a full channel.

`Once`. `Once` is designed to ensure that a global variable is only initialized once. The initialization code can be put into a closure and used as the input parameter of the `call_once()` method of a `Once` object. Even when multiple threads call `call_once()` multiple times, only the first invocation is executed. However, when the input closure of `call_once()` recursively calls `call_once()` of the same `Once` object, a

TABLE 4: How threads communicate. *(Global: global static mutable variables; Sync: the `Sync` trait; O. H.: OS or hardware resources.)*

Software	Unsafe/Interior-Unsafe				Safe		MSG
	Global	**Pointer**	**Sync**	**O. H.**	**Atomic**	**Mutex**	
Servo	1	7	1	0	0	7	2
Tock	0	0	0	2	0	0	0
Ethereum	0	0	0	0	1	2	1
TiKV	0	0	0	1	1	1	0
Redox	1	0	0	2	0	0	0
libraries	1	5	2	0	3	0	0
Total	3	12	3	5	5	10	3

deadlock will be triggered. We have one bug of this type.

Insight 3: *Lacking good understanding in Rust's lifetime rules is a common cause for many blocking bugs.*

Our findings of blocking bugs are unexpected and sometimes in contrast to the design intention of Rust. For example, Rust's automatic unlock is intended to help avoid data races and lock-without-unlock bugs. However, we found that it actually can cause bugs when programmers have some misunderstanding of lifetime in their code.

Suggestion 2: *Future IDEs should add plug-ins to highlight the location of Rust's implicit unlock, which could help Rust developers avoid many blocking bugs.*

Fixing Blocking Bugs. Most of the Rust blocking bugs we collected (51/59) were fixed by adjusting synchronization operations, including adding new operations, removing unnecessary operations, and moving or changing existing operations. One fixing strategy unique to Rust is adjusting the lifetime of the returned variable of `lock()` (or `read()`, `write()`) to change the location of the implicit `unlock()`. This strategy was used for the bug of Figure 8 and 20 other bugs. Adjusting the lifetime of a variable is much harder than moving an explicit `unlock()` as in traditional languages.

The other eight blocking bugs were not fixed by adjusting synchronization mechanisms. For example, one bug was fixed by changing a blocking system call into a non-blocking one.

One strategy to avoid blocking bugs is to explicitly define the boundary of a critical section. Rust allows explicit drop of the return value of `lock()` (by calling `mem::drop()`). We found 11 such usages in our studied applications. Among them, nine cases perform explicit drop to avoid double lock and one case is to avoid acquiring locks in conflicting orders. Although effective, this method is not always convenient, since programmers may want to use `lock()` functions directly without saving their return values (*e.g.*, the `read` lock is used directly at line 3 in Figure 8).

Suggestion 3: *Rust should add an explicit unlock API of `Mutex`, since programmers may not save the return value of `lock()` in a variable and explicitly dropping the return value is sometimes inconvenient.*

5.2 Non-Blocking Bugs

Non-blocking bugs are concurrency bugs where all threads can finish their execution, but with undesired results. This part presents our study on non-blocking bugs.

Rust supports both shared memory and message passing as mechanisms to communicate across threads. Among the 41 non-blocking bugs, three are caused by errors in message passing (*e.g.*, messages in an unexpected order causing programs to misbehave). All the rest are caused by failing to protect shared resources. Since there are only three bugs related to message passing, we mainly focus our study on non-blocking bugs caused by shared memory, unless otherwise specified.

Data Sharing in Buggy Code. Errors during accessing shared data are the root causes for most non-blocking bugs in traditional programming languages [64], [65], [66], [67], [68], [69]. Rust's core safety rules forbid mutable aliasing, which essentially disables mutable sharing across threads. For non-blocking bugs like data races to happen, some data must have been shared and modified. It is important to understand how real buggy Rust programs share data across threads, since differentiating shared variables from local variables can help the development of various bug detection tools [70]. We analyzed how the 38 non-blocking bugs share data and categorized them in Table 4.

Sharing with unsafe code. 23 non-blocking bugs share data using unsafe code, out of which 19 use interior-unsafe functions to share data. Without a detailed understanding of the interior-unsafe functions and their internal unsafe mechanisms, developers may not even be aware of the shared-memory nature when they call these functions.

The most common way to share data is by passing a raw pointer to a memory space (12 in our non-blocking bugs). A thread can store the pointer in a local variable and later dereference it or cast it to a reference. All raw pointer operations are unsafe, although after (unsafe) casting, accesses to the casted reference can be in safe code. Many Rust applications are low-level software. We found the second most common type of data sharing (5) to be accessing OS system calls and hardware resources (through unsafe code). For example, in one bug, multiple threads share the return value of system call `getmntent()`, which is a pointer to a structure describing a file system. The other two unsafe data-sharing methods used in the remaining 6 bugs are accessing static mutable variables which is only allowed in unsafe code, and implementing the unsafe `Sync` trait for a struct.

Sharing with safe code. A value can be shared across threads in safe code[5] if the Rust compiler can statically determine that all threads' accesses to it are within its lifetime and that there can only be one writer at a time. Even though the sharing of any single value in safe code follows Rust's safety rules (*i.e.*, no combination of aliasing and mutability), bugs still happen because of violations to programs' semantics. 15 non-blocking bugs share data with safe code, and we categorize them in two dimensions. To guarantee mutual exclusion, five of them use atomic variables as shared variables, and the other ten bugs wrap shared data using `Mutex` (or `RwLock`). To ensure lifetime covers all usages, nine bugs use `Arc` to wrap shared data and the other six bugs use global variables as shared variables.

Insight 4: *There are patterns of how data is (improperly) shared*

5. Using interior unsafe functions provided by standard libraries is treated as writing safe code here, although those functions contain unsafe code internally.

```
1    impl Engine for AuthorityRound {
2      fn generate_seal(&self) -> Seal {
3 -      if self.proposed.load() { return Seal::None; }
4 -      self.proposed.store(true);
5 -      return Seal::Regular(...);
6 +      if !self.proposed.compare_and_swap(false, true) {
7 +        return Seal::Regular(...);
8 +      }
9 +      return Seal::None;
10     }
11   }
```

Fig. 9: A non-blocking bug in Ethereum.

```
  fn follow_hyperlink(subject: &Element, href: DOMString){
    let doc = document_from_node(subject);
-   let url = base_url(&doc.url()).parse(&href).unwrap();
+   let url = match base_url(&doc.url()).parse(&href) {
+     Ok(url) => url,
+     Err(_) => return,
+   };
    doc.window().load_url(url);
  }
```

Fig. 10: A Panic issue in Servo.

and these patterns are useful when designing bug detection tools.

Bug Analysis. After a good understanding of how Rust programmers share data across threads, we further examine the non-blocking bugs to see how programmers make mistakes. Although there are many unique ways Rust programmers share data, they still make traditional mistakes that cause non-blocking bugs. These include data race [64], [65], [66], atomicity violation [67], [68], [69], and order violation [57], [71], [72], [73].

We examine how shared memory is synchronized for all our studied non-blocking bugs. 17 of them do not synchronize (protect) the shared memory accesses at all, and the memory is shared using unsafe code. This result shows that using unsafe code to bypass Rust compiler checks can severely degrade thread safety of Rust programs. 21 of them synchronize their shared memory accesses, but there are issues in the synchronization. For example, expected atomicity is not achieved or expected access order is violated.

Surprisingly, 25 of our studied non-blocking bugs happen in safe code. This is in contrast to the common belief that safe Rust code can mitigate many concurrency bugs and provide "fearless concurrency" [62], [74].

Insight 5: *How data is shared is not necessarily associated with how non-blocking bugs happen, and the former can be in unsafe code and the latter can be in safe code.*

There are seven bugs involving Rust-unique libraries, including two related to message passing. When multiple threads request mutable references to a `RefCell` at the same time, a runtime panic will be triggered. This is the root cause of four bugs. The `RefCell` objects causing the bugs are shared using the `Sync` trait for two of them and using pointers for the other two. Rust provides a unique strategy where a mutex is poisoned when a thread holding the mutex panics. Another thread waiting for the mutex will receive `Err` from `lock()`. The poisoning mechanism allows panic information to be propagated across threads. One bug is caused by failing to send out a logging message when poisoning happens. The other two bugs are caused by panics when misusing `Arc` or channel. Since these panic issues happen when multiple threads interact with each other, we consider them as concurrency bugs, while the panic issues we will discuss in Section 6 are confined in one single thread.

Insight 6: *Misusing Rust's unique libraries is one major root cause of non-blocking bugs, and all these bugs are captured by runtime checks inside the libraries, demonstrating the effectiveness of Rust's runtime checks.*

Interior Mutability. As explained in Section 2.3, interior mutability is a pattern where a function internally mutates values, but these values look immutable from outside the function. Improper use of interior mutability can cause non-blocking bugs (13 in total in our studied set).

Figure 9 shows one such example. `AuthorityRound` is a struct that implements the `Sync` trait (thus an `AuthorityRound` object can be shared by multiple threads after declared with `Arc`). The `proposed` field is an atomic boolean variable, initialized as `false`. The intention of function `generate_seal()` is to return a `Seal` object only once at a time, and the programmers (improperly) used the `proposed` field at lines 3 and 4 to achieve this goal. When two threads call `generate_seal()` on the same object and both of them finish executing line 3 before executing line 4, both threads will get a `Seal` object as the function's return value, violating the program's intended goal. The patch is to use an atomic instruction at line 6 to replace lines 3 and 4.

In this buggy code, the `generate_seal()` function modifies the immutably borrowed parameter `&self` by changing the value of the `proposed` field. If the function's input parameter is set as `&mut self` (mutable borrow), the Rust compiler would report an error when the invocation of `generate_seal()` happens without holding a lock. In other words, if programmers use mutable borrow, then they would have avoided the bug with the help of the Rust compiler. There are 12 more non-blocking bugs in our collected bug set where the shared object `self` is immutably borrowed by a struct function but is changed inside the function. For six of them, the object (`self`) is shared safely. The Rust compiler would have reported errors if these borrow cases were changed to mutable.

Rust programmers should carefully design interfaces (*e.g.*, mutable borrow vs. immutable borrow) to avoid non-blocking bugs. With proper interfaces, the Rust compiler can enable more checks, which could report potential bugs.

Insight 7: *The design of APIs can heavily impact the Rust compiler's capability of identifying bugs.*

Suggestion 4: *Internal mutual exclusion must be carefully reviewed for interior mutability functions in structs implementing the `Sync` trait.*

Fixes of Non-Blocking Bugs. The fixing strategies of our studied Rust bugs are similar to those in other programming languages [20], [75]. 20 bugs were fixed by enforcing atomic accesses to shared memory. Ten were fixed by enforcing ordering between two shared-memory accesses from different threads. Five were fixed by avoiding (problematic) shared memory accesses. One was fixed by making a local copy of some shared memory. Finally, two were fixed by changing application-specific logic.

Insight 8: *Fixing strategies of Rust non-blocking (and blocking) bugs are similar to traditional languages. Existing automated bug*

TABLE 5: Panic Category. *(Error: missing error handling, Arithmetic: wrong arithmetic operations, Assertion: assertion errors, and OOB: out-of-bounds accesses.)*

Software	Error	Arithmetic	Assertion	OOB	Others
Servo	16	4	3	4	2
Tock	1	1	0	0	0
Ethereum	16	17	6	2	0
TiKV	0	2	3	1	0
Redox	5	4	6	4	1
libraries	1	4	0	1	6
total	39	32	18	12	9

fixing techniques are likely to work on Rust too.

For example, cfix [76] and afix [77] patch order-violation and atomicity-violation bugs. Based on Insight 8, we believe that their basic algorithms can be applied to Rust, and they only need some implementation changes to fit Rust.

6 Unexpected panic issues

The Rust runtime dynamically detects specific types of errors and initiates panics to prevent further damage caused by the errors. While panic issues may have fewer security implications compared with memory bugs, they can abruptly halt programs, resulting in reduced reliability. Additionally, they can potentially be exploited to carry out denial-of-service (DoS) attacks. Therefore, it is critical to comprehend and resolve panic issues in Rust programs. This section presents our study results of 110 programming errors that lead to unexpected panics.

6.1 Bug Analysis Results

We begin by analyzing where the panic issues occur. Unsurprisingly, the majority of these issues (107 out of 110) are in safe code. Although the remaining three are in unsafe code, the same panic issues can happen in safe code. Rust programmers must pay attention to possible panic issues when writing safe code.

Next, we examine the root causes of the panic issues and separate them into five categories. A detailed breakdown of the categories is presented in Table 5.

Missing error handling. As we discussed in Section 2.4, Rust provides `Result` and `Option` enums for callees to either return values (`Ok(T)` or `Some(T)`) or notify their callers of encountered execution errors (`Err(E)` or `None`). Additionally, for improved programming convenience, Rust allows programmers to assume that a callee function always executes successfully and directly retrieve the actual return value using `unwrap()` or `expect()`[6] on the returned `Result` (or `Option`) object. However, if this assumption is incorrect, the `unwrap()` (or `expect()`) call triggers a panic. In total, 39 panic issues stem from this cause, with 17 resulting from misuse of `Option` objects and 22 resulting from misuse of `Result` objects. When considering how programmers access the internal values, 27 issues are triggered by calling `unwrap()`, while the remaining 12 are triggered when invoking `expect()`.

Figure 10 illustrates an example from Servo to exemplify this category. The programmer mistakenly assumed

6. The difference between `expect()` and `unwrap()` is that `expect()` takes a string as its parameter and when it triggers a panic, it uses the string as the panic message.

```
1    impl kernel::SysTick for SysTick {
2      fn set_timer(&self, us: u32) {
3        let tenms: u32 = self.tenms();
4 -      let reload = tenms * us / 10000;
5 +      let reload = tenms / (10000 / us);
6
7        self.regs.value.set(0);
8        self.regs.reload.set(reload);
9      }
10   }
```

Fig. 11: An arithmetic overflow example in Tock. *(The patch is simplified to ease the explanation.)*

the `parse()` function call at line 3 would always return `Ok(Url)`, leading him to call `unwrap()` on the returned `Result` object. However, it is possible for the `parse()` function call to fail and return `Err(ParseError)` instead. As a result, a panic is triggered.

Wrong arithmetic operations. The Rust runtime pinpoints errors when conducting certain types of arithmetic operations and triggers panics accordingly. In total, 32 panic issues arise from computation errors, with 29 of them stemming from overflow. Figure 11 shows one example. At line 4, the programmer multiplies two `u32` numbers (`us` and `tenms`) and divides the result by `10000`. However, the multiplication can overflow and triggers a panic when the program is built in the debug mode. It is important to note that the Rust runtime behaves differently for various numeric types and between the debug and release building modes. In the case of Figure 11, no panic occurs if the program is built in the release mode. Moreover, an overflow in float number computations does not trigger any panic regardless of which building mode is used.

For the remaining three issues, one is caused by dividing by zero, and the other two are due to casting a value to a type, while the value is larger than the maximum value of that type.

Assertion errors. Sometimes, programmers may mark some program execution points as unreachable by calling `panic!()`, `unimplemented!()`, or `unreachable!()`, or enforce some constraints on processed data by calling `assert!()`. When these execution points are reached or the constraints are violated, panics are triggered. In our study, 18 panic issues are caused by this reason.

Out of bounds. For std composite types (*e.g.*, `Vec![T]`, `String`) that implement the `Index` trait, the Rust runtime raises panics when attempting to use an out-of-bounds index to access elements in objects of these types. Among our studied issues, 12 fall into this category.

Others. There are nine panic issues not in the above categories. Among them, seven are caused by stack overflow when triggering infinite recursive function calls, and the other two are due to calling a library function with a wrong parameter.

Insight 9: *Rust offers extensive capabilities for capturing runtime errors and also enables programmers to express and handle possible execution errors by themselves. However, incomplete estimation of the program logic can result in the misuse of these functionalities, causing unexpected panics.*

Suggestion 5: *Future techniques can examine program sites that can trigger panics and investigate whether these sites can be reached to identify possible bugs or verify whether a specific code* *segment or the entire program is panic-free.*

6.2 Fixing Strategies

We categorize the fixing strategies of our collected panic issues into four categories.

Eliminate panic statements. The elimination of the statements triggering panics were employed to fix 37 panic issues. For example, in Figure 11, the panic issue was resolved by replacing the multiplication operation causing the panic in the dividend with a division operation in the divisor. Another example is fixing a panic issue caused by executing an `unimplemented!()` statement. It was patched by replacing the `unimplemented!()` statement with code implementing the desired functionality.

Conditionally skip code. Similar to how some memory bugs were addressed in Section 4.2, modifying branch conditions to selectively skip code sites triggering panics fixed 32 panic issues.

Add error handling. The addition of proper error handling code was utilized to resolve 26 panic issues caused by calling `unwrap()` or `expect()` on a `Result` or `Option` object. For example, the panic issue in Figure 10 was fixed by introducing a `match` block to handle the two possible values of the returned `Result` object. In this case, the execution of function `follow_hyperlink()` terminates earlier (by returning at line 6), when the `Result` object is an `Err(ParseError)`.

Propagate errors. The remaining 15 panic issues were fixed by propagating the errors causing the panics to the caller functions by returning `Result` or `Option` objects, as the caller functions may possess additional contextual information to effectively handle the errors.

Insight 10: *There are common strategies in fixing Rust's panic issues, and some of them are unrelated to program semantics (e.g., propagating errors, eliminating panic statements), indicating the research opportunities in automatically fixing programming errors leading to unexpected panics.*

7 Static Bug Detection

Our empirical bug study reveals that Rust's compiler checks do not adequately cover all types of bugs. We believe that Rust bug detection tools should be developed and our study findings can significantly contribute to these developments. To demonstrate the value and potential of our study results, we have created five static checkers to pinpoint one type of memory bugs, three types of concurrency bugs, and potential locations that can trigger runetime panics. These checkers represent our initial efforts in combating Rust bugs. However, we emphasize that bug detection for Rust should go beyond our current work. We strongly encourage researchers and practitioners to invest further in Rust bug detection based on our exploration.

In this section, we begin by discussing the designs and algorithms employed in our checkers. Subsequently, we present the experimental results, which encompass the number of previously unknown bugs detected by our checkers, our checkers' bug coverage, the number of their reported false positives, and their execution time.

7.1 Design and Algorithm

We construct all our static checking by analyzing Rust's mid-level intermediate representation (MIR) [78]. MIR provides us with a wealth of information, including details about types, def-use chains, and control flows. Furthermore, MIR distinguishes between pointers and references, as well as between mutable and immutable references. Additionally, MIR calls `StorageLive` and `StorageDead` on each variable to mark the start and end of its lifetime, respectively. Below, we describe how we build the six checkers using the information provided by MIR one by one.

Use after free. As we discussed in Section 4.1, use-after-free bugs primarily occur when a pointer is utilized to access an object beyond the object's lifetime scope. Different from LLVM IR, MIR provides a clear distinction between pointers and references, which allows us to concentrate our detection analysis on pointer usages, since Rust's safety checks guarantee that all references are used within objects' lifetime and cannot cause use-after-free bugs.

We begin our analysis by identifying all functions containing pointer usages. We then conduct intra-procedural analysis on each identified function. We implement a points-to analysis to determine which object a pointer points to. For a given object, we consider the instructions between its `StorageLive` and `StorageDead` calls as its lifetime scope, since these two calls mark where the object's lifetime starts and ends, respectively[7]. When the ownership of an object is moved from variable A to variable B, MIR signals the end of A's lifetime (via a `StorageDead` call on A) and the start of B's lifetime (via a `StorageLive` call on B). We check each move operation and merge lifetime scopes of objects involved in a *move-to* relationship. If a dereference of a pointer can be reached from the location where the object the pointer points to terminates its lifetime, we report a use after free. Moreover, if an object is dropped in a function, but a pointer to that object escapes from the function (*e.g.*, returned by the function or saved to a global variable), we also report a use after free.

Double lock. The high-level idea of detecting double locks is straightforward: we inspect each critical section and check whether the same lock is acquired again within that critical section. However, one challenge we face is that MIR does not provide locations where `unlock()` is called. Thus, we must compute lifetime scopes of `LockGuard` variables returned by locking operations to determine the boundaries of critical sections. On the other hand, the `move` mechanism simplifies the computation when a `LockGuard` is moved to a callee, since the `LockGuard` must be dropped in the callee. The critical section ends immediately after the call site in the caller function, and we do not need to rely on inter-procedural analysis to gain this conclusion. Those distinctions set our detector apart from existing double-lock detection techniques built for other languages [79], [80], since those existing techniques can rely on where `unlock()` is called to determine critical section boundaries.

To implement this in detail, we begin by analyzing the input parameter and return value of each locking operation

7. A `StorageDead` call on an object usually follows a `drop` call on the object, which frees the object.

TABLE 6: Benchmarks and Evaluation Results. *(Under the "App Info" columns, LOC denotes lines of source code, and * denotes not applicable. Under the "# of Bugs" columns, UAF denotes use after free, DL denotes double lock, CO denotes conflicting ordering deadlock, AV denotes atomicity violation, x_y denotes x bugs and y false positives, and - denotes both the bug number and the false positive number are zero. Under the "Time" columns, Detection denotes the sum of the execution time of the five bug detectors, Building denotes the building time of an application, Overhead denotes the division of the detection time over the building time, and all execution time is measured in seconds.)*

	App Info			# of Bugs							Time		
Software	Stars	Commits	LOC	**UAF**	**DL**	**CO**	**AV**	**Total**	**Panic**	**Rudra**	**Detection (s)**	**Building (s)**	**Overhead**
Servo	19199	43945	320570	-	0_1	-	1_0	1_1	20682	0_5	71	1207	5.88%
Tock	2695	7986	117038	-	-	-	-	-	802	0_1	4	96	4.17%
Ethereum	6310	12109	128626	-	13_3	14_0	3_0	30_3	1052	-	70	963	7.27%
TiKV	8903	5591	237614	-	-	-	-	-	2477	0_2	329	3696	8.90%
Redox	12966	2434	59045	5_0	-	-	-	4_1	705	1_3	14	211	6.64%
libraries	4838	2784	26546	-	-	-	2_1	2_1	120	0_{23}	1	22	4.55%
Wasmer	8943	9528	75168	1_0	12_0	0_1	-	13_1	1957	0_3	27	267	10.11%
Substrate	3969	5084	247304	-	2_0	-	-	2_0	1403	-	286	5813	4.92%
Solana	1313	13302	214405	-	9_0	8_0	1_0	18_0	3376	0_2	164	4231	3.88%
Lighthouse	955	3925	102936	-	9_0	-	-	9_0	1707	-	139	941	14.77%
Grin	4770	2447	44779	-	1_0	2_0	1_0	4_0	454	-	16	204	7.84%
Winit	1657	2496	29169	-	1_0	-	-	1_0	111	-	6	84	7.14%
Tokio	11213	2377	46834	-	-	-	1_1	1_1	309	0_1	5	65	7.69%
Firecracker	14675	2680	61863	-	-	-	1_0	1_0	402	-	6	58	10.34%
RCore	1559	1878	22013	-	5_0	3_0	-	8_0	178	-	5	64	7.81%
Serenity	1361	2070	30116	-	-	-	-	-	310	-	20	192	10.42%
Diem	15704	8188	258658	-	-	-	1_0	1_0	4604	-	98	1931	5.08%
Deno	73109	5264	56502	-	-	-	-	-	1525	0_1	55	620	8.87%
Total	*	*	*	6_0	52_5	27_0	11_2	96_7	42174	1_{41}	1316	20665	6.37%

to determine which mutex the returned `LockGuard` belongs to. To differentiate between different mutexes, we use the declaration site as the identifier for each local or global mutex variable, and the struct type along with the field offset as the identifier for each struct-field mutex. Next, we perform interprocedural Gen-Kill analysis to compute live `LockGuard` variables for each basic block. If a basic block contains a locking operation, then its Gen set contains the `LockGuard` returned by the operation. If a basic block moves a `LockGuard` from one variable to another, its Gen set contains the `LockGuard` variable that takes the ownership, while its Kill set contains the `LockGuard` variable from which the ownership is transferred. If a basic block moves a `LockGuard` to a callee function, its Gen set contains the `LockGuard`. When a `LockGuard` is dropped (*i.e.*, used as the parameter to call function `drop()`), `unlock()` is implicitly called on the corresponding mutex. In the case where a basic block drops a `LockGuard`, its Kill set contains the `LockGuard`. For all other scenarios, the Gen and Kill sets are all empty. Finally, we revisit each locking operation and check whether any live `LockGuard` shares the same mutex as the locking operation.

Conflicting ordering deadlock. A conflicting ordering deadlock happens when multiple threads are stuck in a circular wait, each waiting for locks held by the others. Theoretically, a conflicting ordering deadlock can involve multiple mutexes. However, all the seven conflicting ordering deadlocks studied in Section 5.1 involve only two mutexes. As a result, we focus our detection efforts on bugs caused by interactions between two mutexes. The high-level algorithm considers there is a locking order from A to B when the analyzed program acquires lock B while already holding lock A. The algorithm identifies a conflicting ordering deadlock when there are locking orders both from A to B and from B to A.

We implement the detector in a similar way as when we detect double locks. Specifically, we track the lifetimes of `LockGuard` variables to figure out when acquired locks are released. We utilize the same method as the double lock detector to distinguish between different mutexes. Moreover, we implement a Gen-Kill algorithm to compute live `LockGuard` variables for each basic block. The different part is that we check each locking operation and live `LockGuard` at the basic block to compute locking orders and detect conflicting ordering deadlocks after having all locking orders of the program.

Atomicity violation. Atomicity violations occur when two consecutive memory accesses to a shared variable are interleaved by an access from a different thread targeting the same variable [68]. When developing the checker, our focus is primarily on atomicity violations resulting from the misuse of atomic instructions. These instructions enable programmers to access shared variables without using unsafe code, and bugs caused by mistakenly using atomic instructions are more likely to be overlooked by programmers. Moreover, we further narrow down our analysis to cases where the two consecutive memory accesses are in the same function. This strategy significantly simplifies our static analysis and also reduces the likelihood of false positives, since interleaving two accesses in separate functions is more likely to be harmless compared with interleaving two accesses within the same function. Although there are four bug patterns for atomicity violations based on whether the memory accesses are reads or writes [68], we specifically concentrate on violations caused by a remote write interleaving a write following a read, as guided by our bug study.

In our detailed implementation, we begin by identifying functions that perform both atomic reads and writes on the same shared variable. Next, for each identified function, we examine if there exists an atomic write operation that is control-dependent on the value read from a previous atomic read of the same variable. If such a condition is met, we report it as a bug.

Panic. We identify code sites that can trigger unexpected panics by detecting certain standard library function calls. These include calling `unwrap()` or `expect()` on a `Result` or `Option` object, as well as invoking three macros `panic!()`, `panic_fmt!()`, and `assert_failed!()`.

7.2 Evaluation

Our experiments seek to comprehend the effectiveness, accuracy, and performance of our checkers. Specifically, we will answer the following three research questions:

- *RQ-1:* Do our checkers identify previously unknown bugs in real-world Rust programs?
- *RQ-2:* How accurate are the detection results of our checkers?
- *RQ-3:* What is the analysis time required by our checkers for a Rust program?

Benchmarks. We utilize our checkers on the most recent versions of the ten Rust software programs discussed in Section 3 (comprising five applications and five libraries) to determine their capability in detecting previously unknown bugs. Additionally, we choose another 12 applications to evaluate whether our checkers are general enough to identify bugs beyond our studied software. All our selected benchmarks fulfill the following three conditions: 1) being popular on GitHub and having many GitHub stars, 2) having a source code size exceeding 10K lines (except some libraries), and 3) being actively maintained by programmers. Table 6 provides detailed information about the benchmarks.

Baseline technique. We compare our checkers with Rudra [30] and MirChecker [24]. Rudra contains three static checkers designed to identify two types of memory bugs and one type of concurrency bugs, respectively. We apply all the three Rudra checkers to our benchmarks and compare the number of bugs they detect with our checkers. MirChecker, like our checkers, is built on MIR analysis. However, it is implemented using an outdated version of Rust, preventing us from running it on the latest benchmark versions. To address this limitation, we downgrade the benchmark programs to their most recent versions before December 29, 2020 (the date of MirChecker's most recent commit). Subsequently, we compare our checkers with MirChecker on these downgraded benchmarks.

Platform. We conducted all of our experiments on an Ubuntu 18.04 machine equipped with an Intel(R) Core(TM) i7-8750H CPU, 16GB RAM memory and a 1TB SSD.

7.2.1 Effectiveness

Overall results. As shown in Table 6, our detectors successfully identify a total of 96 previously unknown bugs. We report all these bugs to the programmers. Up to this point, the programmers have fixed 45 bugs based on our reporting and have confirmed 41 extra bugs as real bugs. The large number of detected bugs serves as a testament to *the good bug detection capability of our checkers.*

Out of the 12 benchmarks that are additionally selected, our checkers detect bugs in ten of them (from row Wasmer to row Deno in Table 6), which demonstrates that *the buggy code patterns covered by our checkers are general enough and bugs following in the patterns are prevalent in Rust programs.*

```
1   unsafe fn fmt_time(...) -> *const c_char {
2     let time_str = format!(...);
3
4 -   time_str[0..26].as_ptr() as _
5 +   let ptr = time_str[0..26].as_ptr() as *const c_char;
6 +   mem::forget(time_str);
7 +   ptr
8   }
```

Fig. 12: A detected use-after-free bug in Wasmer.

Results of individual checkers. As shown by the last row of Table 6, the double-lock detector identifies the highest number of previously unknown bugs, closely followed by the conflicting-ordering-deadlock detector. They detect 52 and 27 bugs, respectively. The large numbers of detected bugs serve as substantial evidence that Rust programmers frequently make mistakes in inferring the release point of a lock, thereby affirming Insight 3 of our bug study (Section 5.1).

The atomicity-violation detector detects 11 bugs. Similar to Figure 9, nine of the detected bugs are in interior mutability functions, validating Insight 7 of our study (Section 5.2).

We find six use-after-free bugs, with one specific example illustrated in Figure 12. In this case, the `format!` macro creates a string called `time_str` at line 2. A pointer pointing to `time_str`'s internal buffer is returned by the function. However, `time_str` is dropped at the end of the function (at the termination of `time_str`'s lifetime), resulting in the freeing of its internal buffer. Consequently, any subsequent use of the returned pointer will lead to a use-after-free bug. The patch is to call `mem::forget()` at line 6 to prevent the automatic deallocation of `time_str`.

Comparing with Rudra. We execute the Rudra checkers on all benchmark projects. Although Rudra reports 42 suspicious code sites (19 reported by the Sync/Send variance checker and 23 reported by the unsafe dataflow checker), upon thorough examination of the results, we determine that there is only one real bug. The bug is an invalid-free bug. It occurs when the Rust compiler deallocates a vector containing uninitialized memory objects after a user-provided function panics. Our tools identify a significantly higher number of bugs than Rudra for two main reasons. First, our tools are designed based on an empirical study, ensuring that the bugs they cover are more likely to occur in real-world Rust programs. Second, the Rudra paper authors scanned all registered Rust crates during their project, resulting in the resolution of many bugs. In summary, our tools cover distinct patterns of buggy code compared to Rudra, and they *effectively complement Rudra in Rust bug detection.*

Comparing with MirChecker. Out of the 22 downgraded benchmarks, we successfully compiled ten of them with the matched Rust compiler versions. However, for the remaining 12 benchmarks, at least one dependency is either broken or no longer available. MirChecker, when applied to the ten compiled benchmarks, reports seven potential memory bugs, but upon further analysis, all of them are found to be false positives. Additionally, MirChecker flags 327 panic issues, including 40 due to assertion errors, 125 due to out-of-bounds accesses, and 162 due to incorrect arithmetic operations. Among them, MirChecker reports two caused by out-of-bounds accesses as proved to occur, while reporting the other 325 as potential panic issues. In contrast, our checkers detected 19 double-lock deadlocks, seven conflicting-ordering deadlocks, and six atomicity violations. Moreover, our tools identified 7259 code sites that have the potential to cause assertion errors. In summary, our tools demonstrate a higher capability in detecting bugs compared to MirChecker.

Identified panic sites. We identify 42174 code sites that have the potential to trigger panics. While not all of these sites necessarily represent real bugs, we strongly encourage programmers to thoroughly inspect these sites, or employ some directed fuzzing techniques [81], [82] to determine whether these sites can really trigger unexpected panics.

7.2.2 Accuracy

Overall results. In total, our checkers report only seven false positives across the 22 benchmark programs. The true-bug-vs-false-positive rate is around 14:1. This signifies that *our checkers possess a high level of accuracy when analyzing Rust programs.*

The false positives are due to two reasons. First, the double-lock detector reports five false positives, which are a result of incorrect alias information used in determining which locking operations create `LockGuard` variables. Second, the atomicity-violation detector reports two false positives. The reason is that our checker only analyzes atomic operations and it ignores other synchronization operations that ensure the atomic execution of the identified read and write instructions.

Comparing with Rudra. Upon analyzing the 22 benchmark programs, Rudra checkers yield a total of 41 false positives. These false positives can be attributed to two primary reasons. First, Rudra's Sync/Send variance checker generates 19 false alarms concerning self-implemented synchronization primitives. Although the primitives violate the Sync/Send rules checked by the checker, they have specific implementations that ensure the absence of concurrency bugs. Second, programmers employ some mechanisms to prevent memory bugs during panic situations. Unfortunately, Rudra's unsafe dataflow checker fails to identify these mechanisms and mistakenly considers them as memory bugs, causing 22 false positives. In comparison, our checkers exhibit significantly higher accuracy than the state-of-the-art Rust bug detention technique.

Comparing with MirChecker. MirChecker reports seven potential memory bugs, all of which, upon further examination, turn out to be false positives. These false positives are associated with the same logging function, which is implemented in safe code and invoked by safe code. Consequently, there is no use after free, as reported by MirChecker. The function has multiple lifetime annotations on its arguments, which might be the source of confusion for MirChecker. In contrast, our checkers do not generate any false positives while detecting memory bugs. It is crucial to underscore that MirChecker raises a potential panic issue if it cannot definitively confirm that a panic will not occur. Out of the 327 panic issues reported by MirChecker, only two, resulting from out-of-bounds accesses, are proven to occur. The remaining 325 panic issues are possible panic issues, aligning with those identified by our checkers.

7.2.3 Performance

To measure the execution time, we conduct ten runs of each checker on every benchmark program and record the average execution time. Subsequently, we sum the execution time of the five checkers for each program, as depicted in the "Detection" column of Table 6. Roughly speaking, our checkers require more time for analyzing larger benchmark programs (programs with a greater number of lines of code). Specifically, our checkers complete their analysis within ten seconds for six programs, all of which have less than 100K lines of code, except for Tock. Furthermore, our checkers spend less than 100 seconds on additional eight programs. For the remaining four programs, our checkers need more than 100 seconds for their analysis. The program that consumes the most time is TiKV, with our checkers taking 329 seconds to analyze it.

As our checkers can function as plugins of the Rust compiler, we further compare the execution time of our checkers with the building time of the benchmark programs. This evaluation aims to determine the overhead our checkers would introduce if integrated into the building process. To measure the building time, we pre-download all the required crates and configure the number of cargo threads to be one. As shown in Table 6, our checkers impose an overhead of more than 10% for four benchmark programs. Notably, Lighthouse experiences the largest overhead, with 14.7% increase. For all the remaining 18 benchmark programs, our checkers introduce an overhead of less than 10%. Moreover, in the case of four benchmark programs, the overhead is less than 5%. For instance, Solana encounters a mere 3.8% overhead, which is the lowest overhead among all programs. Overall, our checkers introduce small overhead, indicating that *they have a good chance to be integrated into the building process.*

8 Related Work

8.1 Bug Detection in Rust

The Rust runtime detects and triggers a panic on certain types of bugs, such as buffer overflow, division by zero and stack overflow. Rust also provides more bug-detecting features in its debug building mode, including detection of double locks and integer overflow . Capturing a bug and reporting a panic using the runtime can avoid the bug from causing a more significant impact (*e.g.*, being exploited), but the bug can potentially be leveraged to conduct DoS attacks and still needs to be fixed. In Section 6, we study real programming errors that can cause unexpected panics.

Rust uses LLVM [83] as its backend. Many static and dynamic bug detection techniques [57], [58], [68], [84] designed for C/C++ can also be applied to Rust. However, it is still valuable to build Rust-specific detectors such as our tools in Section 7, because Rust's new language features and libraries can cause new types of bugs.

Researchers have developed several bug detection techniques for Rust. The static detectors can analyze the entire

Rust program, providing good code coverage, but they tend to produce false positives. Rust-clippy [85] aims to capture memory bugs that follow certain simple source-code patterns. It only covers a limited number of buggy patterns. FFIChecker detects memory bugs resulting from errors in integrating Rust code with C/C++ using FFI [86]. MirChecker [24] computes numeric and symbolic information and then applies constraint solving to detect assertion errors and memory bugs due to using an owner variable after its ownership is moved through unsafe code. Rudra [30] statically bugs in three patterns.

Dynamic detectors reply on user-provided inputs that can trigger bugs, but their detection results are generally more precise than those of static detectors. Miri [87] is a dynamic memory-bug detector that interprets and executes MIR. Jung *et al.* proposed an alias model for Rust [88]. Based on this model, they built a dynamic memory-bug detector that uses a stack to dynamically track all valid references/pointers to each memory location and reports potential undefined behavior and memory bugs when references are not used in a properly-nested manner. Liu *et al.* introduced a separation of heap memory into safe objects accessed only by safe code and unsafe objects potentially accessed by unsafe code. They then dynamically detected memory bugs or prevented memory corruption when an instruction that should access an unsafe object accesses an safe object [89].

These existing Rust bug detection tools all have their own limitations, and they are far from covering all Rust memory bugs and concurrency bugs. An empirical study on Rust bugs like this work is important. It can help future researchers and practitioners to build more Rust-specific detectors. In fact, we have built five bug detectors based on the findings in our study, and those detectors reveal many previously undiscovered bugs.

8.2 Formalizing and Proving Rust's Correctness

Several previous works aim to formalize or prove the correctness of Rust programs [14], [15], [16], [17], [90]. RustBelt [14] conducts the first safety proof for a subset of Rust. Patina [90] proves the safety of Rust's memory management. Baranowski *et al.* extend the SMACK verifier to work on Rust programs [15]. After formalizing Rust's type system in CLP, Rust programs can be generated by solving a constraint satisfaction problem, and the generated programs can then be used to detect bugs in the Rust compiler [16]. K-Rust [17] compares the execution of a Rust program in K-Framework environment with the execution on a real machine to identify inconsistency between Rust's specification and the Rust compiler's implementation.

In contrast to the aforementioned works, our research aims to understand common mistakes made by real Rust developers, and identifies these mistakes through static analysis. It can improve the safety of Rust programs from a practical perspective.

8.3 Empirical Studies

In the past, researchers have conducted various empirical studies on different kinds of bugs in different programming languages [54], [55], [91], [92], [93], [94], [95]. Our study stands out as one of the early investigation into real-world mistakes in Rust programs. Xu *et al.* analyzing all Rust CVEs prior to 2021 to understand Rust's effectiveness in preventing memory bugs and common patterns of Rust memory bugs [96]. Their study shares some common findings as ours (*e.g.*, Insight 1). However, our work goes beyond their scope by not only examining Rust memory bugs but also investigating concurrency bugs and unexpected panic issues in Rust programs, thus revealing more significant insights. In our previous paper, we inspected memory bugs and concurrency bugs in Rust programs [31]. As panics can cause unexpected halts during program execution, we expand upon our previous research by inspecting programming errors that lead to unexpected panics (Section 6).

There are a few empirical studies on Rust's unsafe code usage. One study counts the amount of unsafe code in *crates.io* [97]. Another one analyzes several cases where interior unsafe is not well encapsulated [98]. Sun *et al.* count the number of Rust libraries that depend on external C/C++ libraries [99]. Evans *et al.* compute the portion of functions that possibly execute unsafe code in real-world Rust libraries and also conduct a survey to understand why developers use unsafe code [100]. Our bug study reveals many issues caused by Rust's unsafe code and it complements those existing studies.

9 Conclusion

As a programming language designed for safety, Rust provides a suite of compiler checks to rule out memory-safety and thread-safety issues. Facing the increasing adoption of Rust in mission-critical systems like OSes and browsers, this paper conducts a comprehensive, empirical study on memory bugs, concurrency bugs, and unexpected panics in real-world Rust programs. Many insights and suggestions are provided in our study. To showcase the value of our study, we further build five static detectors by leveraging the observations of our study. These detectors pinpoint 96 previously unknown bugs in real-world Rust programs. We expect our study to deepen the understanding of real-world safety issues in Rust and guide the programming and research tool design of Rust.

References

[1] S. Klabnik and C. Nichols, "The Rust Programming Language," 2018. [Online]. Available: https://doc.rust-lang.org/stable/book/2018-edition/

[2] S. Shanker, "Safe Concurrency with Rust," 2018. [Online]. Available: http://squidarth.com/rc/rust/2018/06/04/rust-concurrency.html

[3] T. R. Team, "Rust Empowering everyone to build reliable and efficient software," 2019. [Online]. Available: https://www.rust-lang.org/

[4] Y. W. Chua, "Appreciating Rust's Memory Safety Guarantees," 2017. [Online]. Available: https://blog.gds-gov.tech/appreciating-rust-memory-safety-438301fee097

[5] B. G. Team, "Rust versus C gcc fastest programs," 2019. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust.html

[6] S. Overflow, "Stack Overflow Developer Survey 2016," 2016. [Online]. Available: https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted

[7] S. Overflow, "Stack Overflow Developer Survey 2017," 2017. [Online]. Available: https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted

[8] S. Overflow, "Stack Overflow Developer Survey 2018," 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted

[9] Servo, "The Servo Browser Engine," 2019. [Online]. Available: https://servo.org/

[10] Quantum, "Quantum," 2019. [Online]. Available: https://wiki.mozilla.org/Quantum

[11] Straits, "Stratis: Easy to use local storage management for Linux," 2019. [Online]. Available: https://stratis-storage.github.io/

[12] Tock, "Tock Embedded Operating System," 2019. [Online]. Available: https://www.tockos.org/

[13] Redox, "The Redox Operating System," 2019. [Online]. Available: https://www.redox-os.org/

[14] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the Rust programming language," in Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '18), Los Angeles, CA, January 2018.

[15] M. Baranowski, S. He, and Z. Rakamarić, "Verifying Rust programs with smack," in Automated Technology for Verification and Analysis (ATVA '18), Los Angeles, CA, Oct. 2018.

[16] K. Dewey, J. Roesch, and B. Hardekopf, "Fuzzing the Rust typechecker using clp (t)," in Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15), Lincoln, NE, Nov. 2015.

[17] S. Kan, D. Sanán, S. Lin, and Y. Liu, "K-rust: An executable formal semantics for Rust," CoRR, 2018.

[18] CVE, "Common Vulnerabilities and Exposures," 2019. [Online]. Available: https://cve.mitre.org/cve/

[19] RustSec, "Security advisory database for Rust crates," 2019. [Online]. Available: https://github.com/RustSec/advisory-db

[20] T. Tu, X. Liu, L. Song, and Y. Zhang, "Understanding real-world concurrency bugs in go," in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), Providence, RI, Apr. 2019.

[21] Z. Cutner, N. Yoshida, and M. Vassor, "Deadlock-free asynchronous message reordering in Rust with multiparty session types," in Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022, pp. 246–261.

[22] D. J. Pearce, "A lightweight formalism for reference lifetimes and borrowing in Rust," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 43, no. 1, pp. 1–73, 2021.

[23] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), 2021, pp. 597–616.

[24] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in Rust programs via static analysis," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), Virtual, 2021.

[25] M. Cui, C. Chen, H. Xu, and Y. Zhou, "SafeDrop: Detecting memory deallocation bugs of Rust programs via static data-flow analysis," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 4, pp. 1–21, 2023.

[26] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "Syrust: automatic testing of Rust libraries with semantic-aware program synthesis," in Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, New York, NY, United States, June 2021, pp. 899–913.

[27] W. Li, D. He, Y. Gui, W. Chen, and J. Xue, "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 (CC '24), Edinburgh, United Kingdom, March 2024, pp. 60–72.

[28] Z. Xu, B. Wu, C. Wen, B. Zhang, S. Qin, and M. He, "RPG: Rust library fuzzing with pool-based fuzz target generation and generic support," April 2024.

[29] Wikipedia, "Lint_(software)," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Lint_(software)

[30] Y. Bae, K. Youngsuk, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in Rust at the ecosystem scale," in Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21), Virtual, 2021.

[31] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20), London, UK, 2020.

[32] S. Overflow, "Stack Overflow Developer Survey 2019," 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

[33] Stack Overflow, "Stack Overflow Developer Survey 2020," 2020. [Online]. Available: https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted

[34] Stack Overflow, "Stack Overflow Developer Survey 2021," 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021

[35] Octoverse, "The State of Octoverse," 2019. [Online]. Available: https://octoverse.github.com/

[36] IotEdge, "IoT Edge Security Daemon," 2019. [Online]. Available: https://github.com/Azure/iotedge/tree/master/edgelet

[37] Firecracker, "Secure and fast microVMs for serverless computing," 2019. [Online]. Available: https://firecracker-microvm.github.io/

[38] K. Boos and L. Zhong, "Theseus: A state spill-free operating system," in Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS '17), Shanghai, China, Oct. 2017.

[39] Tikv, "A distributed transactional key-value database," 2019. [Online]. Available: https://tikv.org/

[40] MSRC, "Why Rust for safe systems programming," 2019. [Online]. Available: https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming

[41] C. Cimpanu, "Microsoft to explore using Rust," 2019. [Online]. Available: https://www.zdnet.com/article/microsoft-to-explore-using-rust

[42] A. O. S. Blog, "AWS' sponsorship of the Rust project," 2019. [Online]. Available: https://aws.amazon.com/blogs/opensource/aws-sponsorship-of-the-rust-project/

[43] R. Amadeo, "Google is now writing low-level Android code in Rust," 2021. [Online]. Available: https://arstechnica.com/gadgets/2021/04/google-is-now-writing-low-level-android-code-in-rust/

[44] Ethereum, "The Ethereum Project," 2019. [Online]. Available: https://www.ethereum.org/

[45] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, "Multiprogramming a 64kb computer safely and efficiently," in Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17), Shanghai, China, October 2017.

[46] Rand, "Rand. A Rust library for random number generation," 2019. [Online]. Available: https://github.com/rust-random/rand

[47] Crossbeam, "Tools for concurrent programming in Rust," 2019. [Online]. Available: https://github.com/crossbeam-rs/crossbeam

[48] Threadpool, "A very simple thread pool for parallel task execution," 2019. [Online]. Available: https://github.com/rust-threadpool/rust-threadpool

[49] Rayon, "A data parallelism library for Rust," 2019. [Online]. Available: https://github.com/rayon-rs/rayon

[50] Lazy-static, "A macro for declaring lazily evaluated statics in Rust."

2019. [Online]. Available: https://github.com/rust-lang-nursery/lazy-static.rs

[51] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards locating a memory corruption vulnerability with your core dump," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), Vienna, Austria, Oct. 2016.

[52] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao, "Pomp: Postmortem program analysis with hardware-enhanced post-crash artifacts," in Proceedings of the 26th USENIX Conference on Security Symposium (Security '17), Vancouver, Canada, Oct. 2017.

[53] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun, "Rept: Reverse debugging of failures in deployed software," in Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '18), Carlsbad, CA, Oct. 2018.

[54] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes – a comprehensive study of real world concurrency bug characteristics," in Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08), Seattle, WA, Mar. 2008.

[55] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16), Atlanta, GA, Apr. 2016.

[56] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, "Jacontebe: A benchmark suite of real-world java concurrency bugs," in 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15), Lincoln, NE, Nov. 2015.

[57] W. Zhang, C. Sun, and S. Lu, "Conmem: detecting severe concurrency bugs through an effect-oriented approach," in Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10), Pittsburgh, PA, Mar. 2010.

[58] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "Conseq: Detecting concurrency bugs through sequential errors," in Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11), New York, NY, USA, 2011.

[59] CVE, "CVE-2018-1000810," 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000810

[60] R. v. Tonder and C. L. Goues, "Static automated program repair for heap properties," in Proceedings of the 40th International Conference on Software Engineering (ICSE '18), Gothenburg, Sweden, May 2018.

[61] S. Huang, J. Guo, S. Li, X. Li, Y. Qi, K. Chow, and J. Huang, "Safecheck: Safety enhancement of java unsafe api," in Proceedings of the 41st International Conference on Software Engineering (ICSE '19), Montreal, Quebec, Canada, May 2019.

[62] Rust-book, "Fearless Concurrency," 2019. [Online]. Available: https://doc.rust-lang.org/book/ch16-00-concurrency.html

[63] R. Martins, "Interior mutability in Rust, part 2: thread safety," 2016. [Online]. Available: https://ricardomartins.cc/2016/06/25/interior-mutability-thread-safety

[64] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10), Vancouver, Canada, Oct. 2010.

[65] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," ACM Transactions on Computer Systems, 15(4):391-411, 1997.

[66] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient detection of data race conditions via adaptive tracking," in Proceedings of the 20th ACM symposium on Operating systems principles (SOSP '05), Brighton, United Kingdom, Oct. 2005.

[67] L. Chew and D. Lie, "Kivati: Fast detection and prevention of atomicity violations," in Proceedings of the 5th European Conference on Computer systems (EuroSys '10), Paris, France, Apr. 2010.

[68] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "Avio: Detecting atomicity violations via access interleaving invariants," in Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06), San Jose, CA, Oct. 2006.

[69] C. Flanagan and S. N. Freund, "Atomizer: A dynamic atomicity checker for multithreaded programs," in Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04), Venice, Italy, Jan. 2004.

[70] J. Huang, "Scalable thread sharing analysis," in Proceedings of the 38th International Conference on Software Engineering (ICSE '16), New York, NY, USA, 2016.

[71] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin, "2nd-strike: Toward manifesting hidden concurrency typestate bugs," in Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11), Newport Beach, CA, Mar. 2011.

[72] B. Lucia and L. Ceze, "Finding concurrency bugs with context-aware communication graphs," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '09), New York, NY, Dec. 2009.

[73] J. Yu and S. Narayanasamy, "A case for an interleaving constrained shared-memory multi-processor," in Proceedings of the 36th annual International symposium on Computer architecture (ISCA '09), Austin, TX, Jun. 2009.

[74] A. Turon, "Fearless Concurrency with Rust," 2015. [Online]. Available: https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[75] H. Liu, Y. Chen, and S. Lu, "Understanding and generating high quality patches for concurrency bugs," in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '16), Seattle, WA, Nov. 2016.

[76] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12), Hollywood, CA, Oct. 2012.

[77] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 11), San Jose, CA, Jun. 2011.

[78] Rust RFC, "1211-mir," 2021. [Online]. Available: https://rust-lang.github.io/rfcs/1211-mir.html

[79] Z. Liu, S. Zhu, B. Qin, H. Chen, and L. Song, "Automatically detecting and fixing concurrency bugs in go software systems," in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), Virtual, USA, 2021.

[80] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," ACM SIGOPS operating systems review, vol. 37, no. 5, pp. 237–252, 2003.

[81] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in Proceedings of the 30th USENIX Security Symposium (USENIX Security '21), Virtual Conference, 2021.

[82] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), Dallas, Texas, USA, 2017.

[83] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in Proceedings of the International Symposium on Code Generation and Optimization (CGO '04'), Washington, DC, USA, 2004.

[84] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08), Berkeley, CA, USA, 2008.

[85] Rust-clippy, "A bunch of lints to catch common mistakes and improve your Rust code," 2019. [Online]. Available: https://github.com/rust-lang/rust-clippy

[86] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting cross-language memory management issues in Rust," in Proceedings of the 27th European Symposium on Research in Computer Security (ESORICS '2022), Copenhagen, Denmark, 2022.

[87] Miri, An interpreter for Rust's mid-level intermediate representation, 2019. [Online]. Available: https://github.com/rust-lang/miri

[88] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, "Stacked borrows: An aliasing model for Rust," in Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '20), New Orleans, LA, January 2020.

[89] P. Liu, G. Zhao, and J. Huang, "Securing unsafe Rust programs with XRust," in Proceedings of the 42nd International Conference on Software Engineering (ICSE '20), Seoul, South Korea, Jul. 2020.

[90] E. Reed, "Patina: A formalization of the Rust programming language," University of Washington, Tech. Rep. UW-CSE-15-03-02, 2015.

[91] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in Proceedings of the 18th ACM symposium on Operating Systems Principles (SOSP '01), Banff, Canada, Oct. 2001.

[92] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13), San Jose, CA, Feb. 2013.

[93] R. Gu, G. Jin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, Aug. 2015.

[94] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 12), Beijing, China, Jun. 2012.

[95] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in Proceedings of the ACM Symposium on Cloud Computing (SOCC' 14), Seattle, WA, Nov. 2014.

[96] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all Rust cves," ACM Trans. Softw. Eng. Methodol., 2021.

[97] A. Ozdemir, "Unsafe in Rust: Syntactic Patterns," 2019. [Online]. Available: https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax

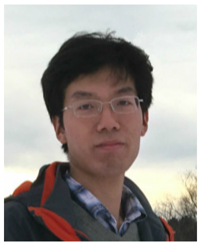
[98] A. Ozdemir, "Unsafe in Rust: The Abstraction Safety Contract and Public Escape," 2019. [Online]. Available: https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-escape

[99] M. Sun, Y. Zhang, and T. Wei, "When memory-safe languages become unsafe," in DEF CON China (DEF CON China '18'), Beijing, China, 2018.

[100] A. N. E. Evans, B. C. Campbell, and M. L. Soffa, "Is Rust used safely by software developers?" in Proceedings of the 42nd International Conference on Software Engineering (ICSE '20), Seoul, South Korea, Jul. 2020.

Boqin Qin received the PhD in computer science from Beijing University of Posts and Telecommunications. He is a researcher and developer in China Telecom Cloud Technology Co., Ltd. He was a visiting student at the Pennsylvania State University and did this work during his visiting. His current research interests focus on program analysis, blockchain security, and distributed systems.

Yilun Chen is a software engineer at HoneycombData Inc. His research interests include operating systems, distributed systems and the reliability of software systems. Chen received his B.Eng from Anhui University and M.S. from the University of Florida.



Haopeng Liu received the PhD degree in computer science from the University of Chicago. His main research interests include program analysis, software systems, and software reliability.

Hua Zhang received the PhD in cryptography from the Beijing University of Posts and Telecommunications. She is an associate professor at the Beijing University of Posts and Telecommunications. Her current research interests focus on smart grids security, network security, cryptographic application, and privacy preserve. She is a Member of IEEE.

Qiaoyan Wen received the PhD degree in cryptography from Xidian University. She is a professor at the Beijing University of Posts and Telecommunications. Her current research interests include cryptography, information security, Internet security, and applied mathematics.

Linhai Song Linhai Song received the PhD in computer science from the University of Wisconsin–Madison. He is an assistant professor in the College of Information Sciences and Technology at the Pennsylvania State University. His main research interests are software reliability and software systems.

Yiying Zhang received the PhD in computer science from the University of Wisconsin–Madison. She is an assistant professor in the Computer Science and Engineering Department at the University of California, San Diego. Her research interests are operating systems, distributed systems, computer architecture, and datacenter networking.