Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs

Boqin Qin*
BUPT, Pennsylvania State University
China, USA

Yilun Chen†
Purdue University
USA

Zeming Yu
Pennsylvania State University
USA

Linhai Song
Pennsylvania State University
USA

Yiying Zhang
University of California, San Diego
USA

Abstract

Rust is a young programming language designed for systems software development. It aims to provide safety guarantees like high-level languages *and* performance efficiency like low-level languages. The core design of Rust is a set of strict safety rules enforced by compile-time checking. To support more low-level controls, Rust allows programmers to bypass these compiler checks to write *unsafe* code.

It is important to understand what safety issues exist in real Rust programs and how Rust safety mechanisms impact programming practices. We performed the first empirical study of Rust by close, manual inspection of 850 unsafe code usages and 170 bugs in five open-source Rust projects, five widely-used Rust libraries, two online security databases, and the Rust standard library. Our study answers three important questions: how and why do programmers write unsafe code, what memory-safety issues real Rust programs have, and what concurrency bugs Rust programmers make. Our study reveals interesting real-world Rust program behaviors and new issues Rust programmers make. Based on our study results, we propose several directions of building Rust bug detectors and built two static bug detectors, both of which revealed previously unknown bugs.

CCS Concepts: • **Software and its engineering → Software safety**; **Software reliability**.

Keywords: Rust; Memory Bug; Concurrency Bug; Bug Study

ACM Reference Format:
Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3386036

1 Introduction

Rust [30] is a programming language designed to build efficient *and* safe low-level software [8, 69, 73, 74]. Its main idea is to inherit most features in C and C's good runtime performance but to rule out C's safety issues with strict compile-time checking. Over the past few years, Rust has gained increasing popularity [46–48], especially in building low-level software like OSes and browsers [55, 59, 68, 71, 77].

The core of Rust's safety mechanisms is the concept of *ownership*. The most basic ownership rule allows each value to have only one *owner* and the value is freed when its owner's *lifetime* ends. Rust extends this basic rule with a set of rules that still guarantee memory and thread safety. For example, the ownership can be *borrowed* or *transferred*, and multiple *aliases* can read a value. These safety rules essentially prohibit the combination of *aliasing* and *mutability*. Rust checks these safety rules at compile time, thus achieving the runtime performance that is on par with unsafe languages like C but with much stronger safety guarantees.

The above safety rules Rust enforces limit programmers' control over low-level resources and are often overkill when delivering safety. To provide more flexibility to programmers, Rust allows programmers to bypass main compiler safety checks by adding an *unsafe* label to their code. A function can be defined as unsafe or a piece of code inside a function can be unsafe. For the latter, the function can be called as a safe function in safe code, which provides a way to encapsulate unsafe code. We call this code pattern *interior unsafe*.

Unfortunately, unsafe code in Rust can lead to safety issues since it bypasses Rust's compiler safety checks. Adding unsafe code and unsafe encapsulation complicates Rust's safety

*The work was done when Boqin Qin was a visiting student at Pennsylvania State University.
†Yilun Chen contributed equally with Boqin Qin in this work.

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386036

semantics. 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? Several recent works [2, 13, 28, 29] formalize and theoretically prove (a subset of) Rust's safety and interior-unsafe mechanisms. However, it is unclear how Rust's language safety and unsafe designs affect real-world Rust developers and what safety issues real Rust software has. With the wider adoption of Rust in systems software in recent years, it is important to answer these questions and understand real-world Rust program behaviors.

In this paper, we conduct the first empirical study of safety practices and safety issues in real-world Rust programs. We examine how safe and unsafe code are used in practice, and how the usages can lead to memory safety issues (*i.e.*, illegal memory accesses) and thread safety issues (*i.e.*, thread synchronization issues like deadlock and race conditions). Our study has a particular focus on how Rust's ownership and lifetime rules impact developers' programming and how the misuse of these rules causes safety issues, since these are Rust's unique and key features.

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 analyzed 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. In total, we studied 850 unsafe code usages, 70 memory-safety issues, and 100 thread-safety issues.

Our study includes three parts. First, we study how unsafe code is used, changed, and encapsulated. We found that unsafe code is extensively used in all of our studied Rust software and it is usually used for good reasons (*e.g.*, performance, code reuse), although programmers also try to reduce unsafe usages when they can. We further found that programmers use interior unsafe as a good practice to encapsulate unsafe code. However, explicitly and properly checking interior unsafe code can be difficult. Sometimes safe encapsulation is achieved by providing correct inputs and environments.

Second, 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 [12] and RustSec [66]. We not only analyze these bugs' behaviors but also understand how the root causes of them are propagated to the effect of them. We found that all memory-safety bugs involve unsafe code, and (surprisingly) most of them also involve safe code. Mistakes are easy to happen when programmers write safe code without the caution of other related code being unsafe. We also found that the scope of *lifetime* in Rust is difficult to reason about, especially when combined with unsafe code, and wrong understanding of *lifetime* causes many memory-safety issues.

Finally, we study concurrency bugs, including non-blocking and blocking bugs [80]. Surprisingly, we found that non-blocking bugs can happen in both unsafe and safe code and that *all* blocking bugs we studied are in safe code. Although many bug patterns in Rust follow traditional concurrency bug patterns (*e.g.*, double lock, atomicity violation), a lot of the concurrency bugs in Rust are caused by programmers' misunderstanding of Rust's (complex) lifetime and safety rules.

For all the above three aspects, we make insightful suggestions to future Rust programmers and language designers. Most of these suggestions can be directly acted on. For example, based on the understanding of real-world Rust usage patterns, we make recommendations on good programming practices; based on our summary of common buggy code patterns and pitfalls, we make concrete suggestions on the design of future Rust bug detectors and programming tools.

With our empirical study results, we conducted an initial exploration on detecting Rust bugs by building two static bug detectors (one for use-after-free bugs and one for double-lock bugs). In total, these detectors found ten previously unknown bugs in our studied Rust applications. These encouraging (initial) results demonstrate the value of our empirical study.

We believe that programmers, researchers, and language designers can use our study results and the concrete, actionable suggestions we made to improve Rust software development (better programming practices, better bug detection tools, and better language designs). Overall, this paper makes the following contributions.

- The first empirical study on real-world Rust program behaviors.
- Analysis of real-world usages of safe, unsafe, and interior-unsafe code, with close inspection of 850 unsafe usages and 130 unsafe removals.
- Close inspection of 70 real Rust memory-safety issues and 100 concurrency bugs.
- 11 insights and 8 suggestions that can help Rust programmers and the future development of Rust.
- Two new Rust bug detectors and recommendations on how to build more Rust bug detectors.

All study results and our bug detectors can be found at https://github.com/system-pclub/rust-study.

2 Background and Related Work

This section gives some background of Rust, including its history, safety (and unsafe) mechanisms, and its current support of bug detection, and overviews research projects on Rust related to ours.

2.1 Language Overview and History

Rust is a type-safe language designed to be both efficient and safe. It was designed for low-level software development

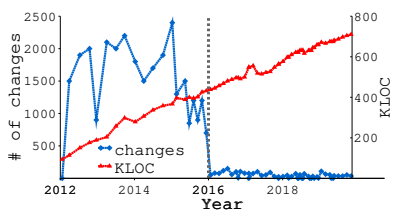

Figure 1. Rust History. *Each blue point shows the number of feature changes in one release version. Each red point shows total LOC in one release version.*

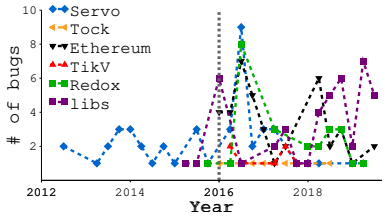

Figure 2. Time of Studied Bugs. *Each point shows the number of our studied bugs that were patched during a three month period.*

Table 1. Studied Applications and Libraries. *The start time, number of stars, and commits on GitHub, total source lines of code, the number of memory safety bugs, blocking bugs, and non-blocking bugs. libraries: maximum values among our studied libraries. There are 22 bugs collected from the two CVE databases.*

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

where programmers desire 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 rule out many potential safety issues. At runtime, Rust behaves like C and could achieve performance that is close to C.

Rust is the most loved language in 2019 according to a Stack Overflow survey [49], and it was ranked as the fifth fastest growing language on GitHub in 2018 [45]. Because of its safety and performance benefits, Rust's adoption in systems software has increased rapidly in recent years [3, 16, 23, 59, 68, 76, 77]. For example, Microsoft is actively exploring Rust as an alternative to C/C++ because of its memory-safety features [9, 44].

Rust was first released in 2012 and is now at version 1.39.0. Figure 1 shows the number of feature changes and LOC over the history of Rust. Rust went through heavy changes in the first four years since its release, and it has been stable since Jan 2016 (v1.6.0). With it being stable for more than three and a half years, we believe that Rust is now mature enough for an empirical study like ours. Figure 2 shows the fixed date of our analyzed bugs. Among the 170 bugs, 145 of them were fixed after 2016. Therefore, we believe our study results reflect the safety issues under stable Rust versions.

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

Figure 3. Sample code to illustrate Rust's safety rules.

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 rejects 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). 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 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, for example, from a caller to a callee and from one thread to another thread. 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.

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. 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. Another extension Rust adds on top of the basic ownership rules is the support of multiple shared read-only references, *i.e.*, immutable references that allow read-only aliasing. A value's reference can also

```
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 }
```

Figure 4. Sample code for (interior) unsafe.

be *mutable*, allowing write access to the value, but there can only be one mutable reference and no immutable references at any single time. After borrowing a value's ownership through *mutable reference*, the temporary owner has the 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 desires to manage safety by themselves (see Section 4 for real examples). Rust allows programs to bypass its safety checking with the *unsafe* feature, denoted by the keyword `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` (Rust traits are similar to interfaces in traditional languages like Java). 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. Figure 4 shows the implementation of a simple struct, which implements the unsafe `Sync` trait at line 2. 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).

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*. Function `set()` in Figure 4 is an interior mutability function. Its input `self` is borrowed immutably, but the `value` field of `self` is changed through pointer `p` (an alias) at line 6.

Many APIs provided by the Rust standard library are interior-unsafe functions, such as `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, and `RwLock`. Section 4.3 presents our analysis of interior unsafe usages in the Rust standard library.

2.4 Bug Detection in Rust

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 build mode, including detection of double lock and integer overflow. These dynamic detection mechanisms Rust provides only capture a small number of issues.

Rust uses LLVM [32] as its backend. Many static and dynamic bug detection techniques [4, 40, 88, 89] designed for C/C++ can also be applied to Rust. However, it is still valuable to build Rust-specific detectors, because Rust's new language features and libraries can cause new types of bugs as evidenced by our study.

Researchers have designed a few bug detection techniques for Rust. Rust-clippy [64] is a static detector for memory bugs that follow certain simple source-code patterns. It only covers a small amount of buggy patterns. Miri [43] is a dynamic memory-bug detector that interprets and executes Rust's mid-level intermediate representation (*MIR*). Jung *et al.* proposed an alias model for Rust [27]. 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. The two dynamic detectors rely on user-provided inputs that can trigger memory bugs. From our experiments, Miri also generates many false positives.

These existing Rust bug detection tools all have their own limitations, and none of them targets 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 two detectors based on our findings in this study, both of which reveal previously undiscovered bugs.

2.5 Formalizing and Proving Rust's Correctness

Several previous works aim to formalize or prove the correctness of Rust programs [2, 13, 28, 29, 61]. RustBelt [28] conducts the first safety proof for a subset of Rust. Patina [61] proves the safety of Rust's memory management. Baranowski *et al.* extend the SMACK verifier to work on Rust programs [2]. 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 [13]. K-Rust [29] 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. Different from these works, our study aims

to understand common mistakes made by real Rust developers, and it can improve the safety of Rust programs from a practical perspective.

2.6 Empirical Studies

In the past, researchers have conducted various empirical studies on different kinds of bugs in different programming languages [7, 19, 20, 24, 34, 38, 39]. As far as we know, we are the first study on real-world mistakes of Rust code.

There are only a few empirical studies on Rust's unsafe code usage similar to what we performed in Section 4. However, the scales of these studies are small on both the applications studied and the features studied. One previous study counts the number of Rust libraries that depend on external C/C++ libraries [72]. One study counts the amount of unsafe code in *crates.io* [50]. Another analyzes several cases where interior unsafe is not well encapsulated [51]. Our work is the first large-scale, systematic empirical study on unsafe in Rust. We study many aspects not covered by previous works.

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 software like ours is important for several reasons. It can demonstrate how real programmers use Rust and how their behavior changes over time. It can also reveal what mistakes (bugs) real programmers make and how they fix them. Some of these usage patterns and 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 write their code 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 [68] 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* [76] is a key-value store that supports both single key-value-pair and transactional key-value accesses. *Parity Ethereum* [15] is a fast, secure blockchain client written in Rust (we call it *Ethereum* for brevity in the rest of the paper). *Redox* [59] is an open-source secure OS that adopts microkernel architecture but exposes UNIX-like interface. *Tock* [35] 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 (also written in Rust). They include 1) *Rand* [56], a library for random number generation, 2) *Crossbeam* [10], a framework for building lock-free concurrent data structures, 3) *Threadpool* [75], Rust's implementation of thread pool, 4) *Rayon* [58], a library for parallel computing, and 5) *Lazy_static* [33], a library for defining lazily evaluated static variables.

Collecting and studying bugs. To collect bugs, we analyzed GitHub commit logs from applications in Table 1. We first filtered the commit logs using a set of safety-related keywords, *e.g.*, "use-after-free" for memory bugs, "deadlock" for concurrency bugs. These keywords either cover important issues in the research community [11, 82, 83] or are used in previous works to collect bugs [34, 36, 39, 80]. We then manually inspected filtered logs to identify bugs. For our memory-safety study, we also analyzed all Rust-related vulnerabilities in two online vulnerability databases, CVE [12] and RustSec [66]. In total, we studied 70 memory and 100 concurrency bugs.

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 Unsafe Usages

There is a fair amount of unsafe code in Rust programs. We found 4990 unsafe usages in our studied applications in Table 1, including 3665 unsafe code regions, 1302 unsafe functions, and 23 unsafe traits. In Rust's standard library (Rust *std* for short), we found 1581 unsafe code regions, 861 unsafe functions, and 12 unsafe traits.

Since `unsafe` code bypasses compiler safety checks, understanding unsafe usages is an important aspect of studying safety practice in reality. We randomly select 600 unsafe usages from our studied applications, including 400 interior

unsafe usages and 200 unsafe functions. We also studied 250 interior unsafe usages in Rust *std*. We manually inspect these unsafe usages to understand 1) why unsafe is used in the latest program versions, 2) how unsafe is removed during software evolution, and 3) how interior unsafe is encapsulated.

4.1 Reasons of Usage

To understand how and why programmers write unsafe code, we first inspect the type of operations these unsafe usages are performing. Most of them (66%) are for (unsafe) memory operations, such as raw pointer manipulation and type casting. Calling unsafe functions counts for 29% of the total unsafe usages. Most of these calls are made to unsafe functions programmers write themselves and functions written in other languages. In Rust *std*, the heaviest unsafe usages appear in the *sys* module, likely because it interacts more with low-level systems.

To understand the reasons why programmers use unsafe code, we further analyze the *purposes* of our studied 600 unsafe usages. The most common purpose of the unsafe usages is to reuse existing code (42%), for example, to convert a C-style array to Rust's variable-size array (called *slice*), to call functions from external libraries like *glibc*. Another common purpose of using unsafe code is to improve performance (22%). We wrote simple tests to evaluate the performance difference between some of the unsafe and safe code that can deliver the same functionalities. Our experiments show that unsafe memory copy with `ptr::copy_nonoverlapping()` is 23% faster than the `slice::copy_from_slice()` in some cases. Unsafe memory access with `slice::get_unchecked()` is 4-5× faster than the safe memory access with boundary checking. Traversing an array by pointer computing (`ptr::offset()`) and dereferencing is also 4-5× faster than the safe array access with boundary checking. The remaining unsafe usages include bypassing Rust's safety rules to share data across threads (14%) and other types of Rust compiler check bypassing.

One interesting finding is that sometimes removing `unsafe` will not cause any compile errors (32 or 5% of the unsafe usages in the applications we studied). For 21 of them, programmers mark a function as unsafe for code consistency (*e.g.*, the same function for a different platform is unsafe). For the rest, programmers use unsafe to give a warning of possible dangers in using this function.

Worth noticing is a special case of using unsafe for warning purposes. Five unsafe usages among the above no-compile-error cases are for labeling struct constructors (there are also 50 such usages in the Rust *std* library). These constructors only contain safe operations (*e.g.*, initializing struct fields using input parameters), but other functions in the struct can perform unsafe operations and their safety depends on safe initialization of the struct. For example, in Rust *std*, the `String` struct has a constructor function `String::from_utf8_unchecked()` which creates a `String` using the input array of characters. This constructor is marked as an unsafe function although the operations in it are all safe. However, other member functions of `String` that use the array content could potentially have safety issues due to invalid UTF-8 characters. Instead of marking all these functions unsafe and requiring programmers to properly check safe conditions when using them, it is more efficient and reliable to mark only the constructor as unsafe. This design pattern essentially *encapsulates* the unsafe nature in a much smaller scope. Similar usages also happen in applications and explained by developers as good practice [78].

Insight 1: *Most unsafe usages are for good or unavoidable reasons, indicating that Rust's rule checks are sometimes too strict and that it is useful to provide an alternative way to escape these checks.*

Suggestion 1: *Programmers should try to find the source of unsafety and only export that piece of code as an* `unsafe` *interface to minimize* `unsafe` *interfaces and to reduce code inspection efforts.*

4.2 Unsafe Removal

Although most of the unsafe usages we found are for good reasons, programmers sometimes remove unsafe code or change them to safe ones. We analyzed 108 randomly selected commit logs that contain cases where `unsafe` is removed (130 cases in total). The purposes of these unsafe code removals include improving memory safety (61%), better code structure (24%), improving thread safety (10%), bug fixing (3%), and removing unnecessary usages (2%).

Among our analyzed commit logs, 43 cases completely change unsafe code to safe code. The remaining cases change unsafe code to interior unsafe code, with 48 interior unsafe functions in Rust *std*, 29 self-implemented interior unsafe functions, and 10 third-party interior unsafe functions. By encapsulating unsafe code in an interior unsafe function that can be safely called at many places, programmers only need to ensure the safety in this one interior unsafe function (*e.g.*, by checking conditions) instead of performing similar checks at every usage of the unsafe code.

Insight 2: *Interior unsafe is a good way to encapsulate unsafe code.*

Suggestion 2: *Rust developers should first try to properly encapsulate unsafe code in interior unsafe functions before exposing them as unsafe.*

4.3 Encapsulating Interior Unsafe

With unsafe code being an essential part of Rust software, it is important to know what are the good practices when writing unsafe code. From our analysis results above and from Rustonomicon [65], encapsulating unsafe code with interior

```
1 impl<T, ...> Queue<T, ...> {
2     pub fn pop(&self) -> Option<T> { unsafe {...}}
3     pub fn peek(&self) -> Option<&mut T> { unsafe {...}}
4 }
5 // let e = Q.peek().unwrap();
6 // {Q.pop()}
7 // println!("{}", *e); < - use after free
```

Figure 5. An interior mutability example in Rust *std*.

unsafe functions is a good practice. But it is important to understand how to properly write such encapsulation.

To answer this question, we first analyze how Rust *std* encapsulates interior unsafe code (by both understanding the code and reading its comments and documentation). Rust std interior unsafe functions are called heavily by Rust software. It is important to both learn from how *std* encapsulates unsafe code and examine if there is any issue in such encapsulation.

In total, we sampled 250 interior unsafe functions in Rust *std*. For the unsafe code to work properly, different types of conditions need to be satisfied. For example, 69% of interior unsafe code regions require valid memory space or valid UTF-8 characters. 15% require conditions in lifetime or ownership.

We then examined how Rust *std* ensures that these conditions are met. Surprisingly, Rust *std* does not perform any explicit condition checking in most of its interior unsafe functions (58%). Instead, it ensures that the input or the environment that the interior unsafe code executes with is safe. For example, the unsafe function `Arc::from_raw()` always takes input from the return of `Arc::into_raw()` in all sampled interior unsafe functions. Rust *std* performs explicit checking for the rest of the interior unsafe functions, *e.g.*, by confirming that an index is within the memory boundary.

After understanding *std* interior unsafe functions, we inspect 400 sampled interior unsafe functions in our studied applications. We have similar findings from these application-written interior unsafe functions.

Worth noticing is that we identified 19 cases where interior unsafe code is improperly encapsulated, including five from the *std* and 14 from the applications. Although they have not caused any real bugs in the applications we studied, they may potentially cause safety issues if they are not used properly. Four of them do not perform any checking of return values from external library function calls. Four directly dereference input parameters or use them directly as indices to access memory without any boundary checking. Other cases include not checking the validity of function pointers, using type casting to change objects' lifetime to static, and potentially accessing uninitialized memory.

Of particular interest are two bad practices that lead to potential problems. They are illustrated in Figure 5. Function `peek()` returns a reference of the object at the head of a queue, and `pop()` pops (removes) the head object from the queue. A use-after-free error may happen with the following sequence of operations (all safe code): a program first calls `peek()` and saves the returned reference at line 5, then calls `pop()` and drops the returned object at line 6, and finally uses the previously saved reference to access the (dropped) object at line 7. This potential error is caused by holding an immutable reference while changing the underlying object. This operation is allowed by Rust because both functions take an immutable reference `&self` as input. When these functions are called, the ownership of the queue is immutably borrowed to both functions.

Table 2. Memory Bugs Category. *Buffer: Buffer overflow; Null: Null pointer dereferencing; Uninitialized: Read 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

According to the program semantics, `pop()` actually changes the immutably borrowed queue. This *interior mutability* (defined in Section 2.3) is improperly written, which results in the potential error. An easy way to avoid this error is to change the input parameter of `pop()` to `&mut self`. When a queue is immutably borrowed by `peek()` at line 5, the borrowing does not end until line 7, since the default lifetime rule extends the lifetime of `&self` to the lifetime of the returned reference [63]. After the change, the Rust compiler will not allow the mutable borrow by `pop()` at line 6.

Insight 3: *Some safety conditions of unsafe code are difficult to check. Interior unsafe functions often rely on the preparation of correct inputs and/or execution environments for their internal unsafe code to be safe.*

Suggestion 3: *If a function's safety depends on how it is used, then it is better marked as unsafe not interior unsafe.*

Suggestion 4: *Interior mutability can potentially violate Rust's ownership borrowing safety rules, and Rust developers should restrict its usages and check all possible safety violations, especially when an interior mutability function returns a reference. We also suggest Rust designers differentiate interior mutability from real immutable functions.*

5 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.

5.1 Bug Analysis Results

It is important to understand both the *cause* and the *effect* of memory-safety issues (bugs). We categorize our studied

```
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 }
```

Figure 6. An invalid-free bug in Redox.

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 [88, 89], 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 functions, their input parameters are used directly or indirectly as an index to access a buffer, without any boundary checks.

Null pointer dereferencing. 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 as the good practices in Section 4.3.

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

```
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 }
```

Figure 7. A use-after-free bug in RustSec.

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 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 and the object will be dropped there. The use of `p` is thus after the object has been freed. 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 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 4: *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 5: *Future memory bug detectors can ignore safe code that is unrelated to unsafe code to reduce false positives and to improve execution efficiency.*

5.2 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 5: *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 [22, 79] that synthesize conditions for dangerous operations to fix Rust memory bugs.

6 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. Similar to a recent work's taxonomy of concurrency bugs [80], we divide our 100 collected concurrency bugs into blocking bugs (*e.g.*, deadlock) and non-blocking bugs (*e.g.*, data race).

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

6.1 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.

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 [42]. 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 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 variable holding the reference 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

```
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 }
```

Figure 8. A double-lock bug in TiKV.

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 reference-holding object returned by `client.read()` spans the whole `match` code block and 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 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 deadlock will be triggered. We have one bug of this type.

Insight 6: *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-unlocks bugs. However, we found that it actually can cause bugs when programmers have some misunderstanding of lifetime in their code.

Suggestion 6: *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).

Table 4. How threads communicate. *Global: global static mutable integer; 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

Suggestion 7: *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.*

6.2 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 [6, 14, 17, 40, 67, 86]. 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 [21]. 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 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 7: *There are patterns of how data is (improperly) shared 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 [14, 67, 86], atomicity violation [6, 17, 40], and order violation [18, 41, 85, 89].

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, 81].

Insight 8: *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. A buggy `RefCell` is shared using the Sync trait

```
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 }
```

Figure 9. A non-blocking bug in Ethereum.

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.

Insight 9: *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 10: *The design of APIs can heavily impact the Rust compiler's capability of identifying bugs.*

Suggestion 8: *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 [37, 80]. 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 11: *Fixing strategies of Rust non-blocking (and blocking) bugs are similar to traditional languages. Existing automated bug fixing techniques are likely to work on Rust too.*

For example, cfix [26] and afix [25] patch order-violation and atomicity-violation bugs. Based on Insight 11, we believe that their basic algorithms can be applied to Rust, and they only need some implementation changes to fit Rust.

7 Bug Detection

Our empirical bug study reveals that Rust's compiler checks fail to cover many types of bugs. We believe that Rust bug detection tools should be developed and our study results can greatly help these developments. Unfortunately, existing Rust bug detectors (Section 2.4) are far from sufficient. From our experiments, the Rust-clippy detector failed to detect any of our collected bugs. The Miri dynamic detector can only report bugs when a test run happens to trigger problematic execution. Moreover, it has many false positive bug reports. Rust's debug mode can help detect double locks and integer overflows. But similar to Miri, it also needs a test input to trigger buggy execution.

This section discusses how our bug study results can be used to develop bug detection tools. We also present two new bug detectors we built for statically detecting Rust use-after-free and double-lock bugs. Note that although the results of these bug detectors are promising, they are just our initial efforts in building Rust bug detectors. Our intention of them is to demonstrate the value and potential of our study results, and they are far from perfect. We encourage researchers and practitioners to invest more on Rust bug detection based on our initial results.

7.1 Detecting Memory Bugs

From our study of memory bugs, we found that many memory-related issues are caused by misuse of ownership and lifetime.

Thus, an efficient way to avoid or detect memory bugs in Rust is to analyze object ownership and lifetime.

IDE tools. Misunderstanding Rust's ownership and lifetime rules is common (because of the complexity of these rules when used in real-world software), and it is the main cause of memory bugs. Being able to visualize objects' lifetime and owner(s) during programming time could largely help Rust programmers *avoid* memory bugs. An effective way of visualization is to add plug-ins to IDE tools, for example, by highlighting a variable's lifetime scope when the mouse/cursor hops over it or its pointer/reference. Programmers can easily notice errors when a pointer's usage is outside the lifetime of the object it points to and avoid a use-after-free bug. Highlighting and annotating ownership operations can also help programmers avoid various memory bugs such as double-free bugs and invalid-free bugs (*e.g.*, Figure 6).

Static detectors. Ownership/lifetime information can also be used to statically detect memory bugs. Based on our study results in Section 5, it is feasible to build static checkers to detect invalid-free, use-after-free, double-free memory bugs by analyzing object lifetime and ownership relationships. For example, at the end of an object's lifetime, we can examine whether or not this object has been correctly initialized to detect invalid-free bugs like Figure 6.

As a more concrete example, we have built a new static checker based on lifetime/ownership analysis of Rust's mid-level intermediate representation (MIR) to detect use-after-free bugs like Figure 7. We chose MIR to perform the analysis because it provides explicit ownership/lifetime information and rich type information. Our detector maintains the state of each variable (*alive* or *dead*) by monitoring when MIR calls `StorageLive` or `StorageDead` on the variable. For each pointer/reference, we conduct a "points-to" analysis to maintain which variable it points to/references. This points-to analysis also includes cases where the ownership of a variable is moved. When a pointer/reference is dereferenced, our tool checks if the object it points to is dead and reports a bug if so.

In total, our detector found four previously unknown bugs [60] in our studied applications. Our tool currently reports three false positives, all caused by our current (unoptimized) way of performing inter-procedural analysis. We leave improving inter-procedural analysis to future work.

Overall, the results of our initial efforts in building static bug detectors are encouraging, and they confirm that ownership/lifetime information is useful in detecting memory bugs that cannot be reported by the Rust compiler.

Dynamic detectors. Apart from IDE tools and static bug detectors, our study results can also be used to build or improve dynamic bug detectors. Fuzzing is a widely-used method to detect memory bugs in traditional programming languages [1, 5, 57, 70, 84, 87]. Our study in Section 5 finds that all Rust memory bugs (after the Rust language has been stabilized in 2016) involve unsafe code (Insight 4). Instead of blindly fuzzing all code, we recommend future fuzzing tools to focus on unsafe code and its related safe code to improve their performance (Suggestion 5).

7.2 Detecting Concurrency Bugs

There are many ways to use our study results of concurrency bugs in Section 6 to build concurrency bug detectors. We now discuss how our results can be used and a real static double-lock detector we built.

IDE tools. Rust's implicit lock release is the cause of several types of blocking bugs such as the double-lock bug in Figure 8 (Insight 6). An effective way to avoid these bugs is to visualize critical sections (Suggestion 6). The boundary of a critical section can be determined by analyzing the lifetime of the return of function `lock()`. Highlighting blocking operations such as `lock()` and channel-receive inside a critical section is also a good way to help programmers avoid blocking bugs.

IDE tools can also help avoid non-blocking bugs. For example, to avoid bugs caused by improper use of interior mutability, we can annotate the call sites of interior mutability functions and remind developers that these functions will change their immutably borrowed parameters. Developers can then closely inspect these functions (Suggestion 8).

Static detectors. Lifetime and ownership information can be used to statically detect blocking bugs. We built a double-lock detector by analyzing lock lifetime. It first identifies all call sites of `lock()` and extracts two pieces of information from each call site: the lock being acquired and the variable being used to save the return value. As Rust implicitly releases the lock when the lifetime of this variable ends, our tool will record this release time. We then check whether or not the same lock is acquired before this time, and report a double-lock bug if so. Our check covers the case where two lock acquisitions are in different functions by performing inter-procedural analysis. Our detector has identified six previously unknown double-lock bugs [52–54] in our studied applications (and no false positives). They have been fixed by developers after we reported them.

Ownership information can also help static detection of non-blocking bugs. For example, for bugs caused by misuse of interior mutability like the one in Figure 9, we could perform the following static check. When a `struct` is sharable (*e.g.*, implementing the `Sync` trait) and has a method immutably borrowing `self`, we can analyze whether `self` is modified in the method and whether the modification is unsynchronized. If so, we can report a potential bug. On the other hand, we do not need to analyze a function that mutably borrows `self`, since the Rust compiler will enforce the function to be called with a lock, guaranteeing the proper synchronization of its internal operations (Insight 10).

Dynamic detectors. Dynamic concurrency-bug detectors often need to track shared variable accesses. Being able to differentiate thread-local variables from shared variables can help lower memory and runtime overhead of dynamic bug detectors' tracking. Our empirical study results identify the (limited) code patterns of data sharing across threads in Rust (Insight 7). Dynamic detectors can use these patterns to reduce the amount of tracking to only shared variables.

We also find that misusing Rust libraries is a major cause of non-blocking bugs, which can be effectively caught by the Rust runtime (Insight 9). Thus, future dynamic tools can focus on generating inputs to trigger such misuses and leverage Rust's runtime checks to find bugs.

8 Discussion

After presenting our study results, findings, insights, and suggestions, we now briefly discuss further implications of our work and some of our final thoughts.

Safe and unsafe code. Most of this study and previous theoretical works [27, 28] center around Rust's safe and unsafe code and their interaction. We use our understanding from this work to provide some hints to answer some key questions in this regard.

1) When and why to use unsafe code? Minimizing the use of unsafe code is generally considered as a good Rust programming practice, as confirmed by our study. However, there can be good reasons to use unsafe code. First, there are many external libraries and certain unsafe *std* library functions that do not readily have their safe versions. Instead of writing new safe code, programmers may want to directly use existing unsafe functions. Second, there are certain low-level hardware and OS interfaces that can only be accessed by unsafe code. Third, programmers may want to write unsafe code for performance reasons. Finally, our study shows that programmers sometimes want to label some code unsafe to maintain code consistency across different versions or to label a small piece of code as unsafe to prevent labeling future unsafe behavior that can happen at more code pieces.

2) How to properly encapsulate unsafe operations? In cases where programmers have to or desire to write unsafe code, we suggest them to take extra care to encapsulate their unsafe code. We believe that proper encapsulation of unsafe operations should come from guaranteeing *preconditions* and *postconditions*. Different unsafe operations have different conditions to meet. Memory-related unsafe operations should have precondition checks that ensure the validity of memory spaces (*e.g.*, pointing to valid addresses) and meet ownership requirements (*e.g.*, memory space not owned by others). Concurrency-related unsafe operations should meet traditional synchronization conditions. Calling unsafe functions like external non-Rust libraries should meet both preconditions (checking inputs) and postconditions (checking return values). One difficulty in encapsulating unsafe operations is that not all conditions can be easily checked. They call for more advanced bug detection and verification techniques.

3) How to change unsafe code to safe code? Depending on the type of unsafe operations, there are various ways to change unsafe code to safe code. For example, Rust's implementation can replace (unsafe) calls to external non-Rust libraries. Atomic instructions can replace (unsafe) read/write to shared variables. Certain Rust *std* safe functions can replace unsafe *std* functions.

Language and tool improvement. Although Rust has already gone through years of evolution, our study still hints at a few Rust language features that could be revisited and improved. For example, we found that Rust's implicit unlock feature is directly related to many blocking bugs. Although its intention is to ease programming and improve safety, we believe that changing it to explicit unlock or other critical-section indicators can avoid many blocking bugs.

More important, our study urges (and helps with) the development of tools to help prevent or detect bugs in Rust. We suggest two directions of tool building. The first is to add IDE plugins to hint Rust programmers about potential bug risks and visualize code scope that requires special attention. For example, an alternative method to the above unlock problem is to keep the implicit unlock feature but add IDE plugins to automatically highlight the scope of critical sections. The second is to build bug detection tools that are tailored for Rust. Our bug study results and the two detectors we built are a good starting point for this direction.

Values beyond Rust. Our study can benefit the community of other languages in several ways. First, other languages that adopt the same designs as Rust can directly benefit from our study. For example, languages like Kotlin [31] also use lifetime to manage resources and can benefit from our lifetime-related study results. Similarly, languages that have ownership features (*e.g.*, C++11 `std::unique_ptr`) can benefit from our ownership-related results. Second, Rust adopts several radical language designs. Our study shows how programmers adapt (or fail to adapt) to these new concepts over time. Future language designers can learn from Rust's success and issues when designing new languages.

9 Conclusion

As a programming language designed for safety, Rust provides a suite of compiler checks to rule out memory and thread safety issues. Facing the increasing adoption of Rust in mission-critical systems like OSes and browsers, this paper conducts the first comprehensive, empirical study on unsafe usages, memory bugs and concurrency bugs in real-world Rust programs. Many insights and suggestions are provided in our study. We expect our study to deepen the understanding of real-world safety practices and safety issues in Rust and guide the programming and research tool design of Rust.

References

[1] AFL. 2019. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[2] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis (ATVA '18)*. Los Angeles, CA.

[3] Kevin Boos and Lin Zhong. 2017. Theseus: A State Spill-free Operating System. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems (PLOS '17)*. Shanghai, China.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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.

[5] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. *Proceedings of the 39th IEEE Symposiums on Security and Privacy (Oakland '18)* (2018).

[6] Lee Chew and David Lie. 2010. Kivati: Fast Detection and Prevention of Atomicity Violations. In *Proceedings of the 5th European Conference on Computer systems (EuroSys '10)*. Paris, France.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the 18th ACM symposium on Operating Systems Principles (SOSP '01)*. Banff, Canada.

[8] Yong Wen Chua. 2017. Appreciating Rust's Memory Safety Guarantees. https://blog.gds-gov.tech/appreciating-rust-memory-safety-438301fee097

[9] Catalin Cimpanu. 2019. Microsoft to explore using Rust. https://www.zdnet.com/article/microsoft-to-explore-using-rust

[10] Crossbeam. 2019. Tools for concurrent programming in Rust. https://github.com/crossbeam-rs/crossbeam

[11] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. 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.

[12] CVE. 2019. Common Vulnerabilities and Exposures. https://cve.mitre.org/cve/

[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. 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.

[14] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*. Vancouver, Canada.

[15] Ethereum. 2019. The Ethereum Project. https://www.ethereum.org/

[16] Firecracker. 2019. Secure and fast microVMs for serverless computing. https://firecracker-microvm.github.io/

[17] Cormac Flanagan and Stephen N Freund. 2004. 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.

[18] Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2011. 2ndStrike: 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.

[19] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. 2015. What Change History Tells Us About Thread Synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo, Italy.

[20] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. 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.

[21] Jeff Huang. 2016. Scalable Thread Sharing Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. New York, NY, USA.

[22] Shiyou Huang, Jianmei Guo, Sanhong Li, Xiang Li, Yumin Qi, Kingsum Chow, and Jeff Huang. 2019. SafeCheck: Safety Enhancement of Java Unsafe API. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. Montreal, Quebec, Canada.

[23] IotEdge. 2019. IoT Edge Security Daemon. https://github.com/Azure/iotedge/tree/master/edgelet

[24] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. 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.

[25] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 11)*. San Jose, CA.

[26] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. Hollywood, CA.

[27] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. 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.

[28] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. 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.

[29] Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 2018. K-Rust: An Executable Formal Semantics for Rust. *CoRR* (2018).

[30] Steve Klabnik and Carol Nichols. 2018. The Rust Programming Language. https://doc.rust-lang.org/stable/book/2018-edition/

[31] Kotlin. 2019. The Kotlin Language. https://kotlinlang.org/

[32] Chris Lattner and Vikram Adve. 2004. 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.

[33] Lazy-static. 2019. A macro for declaring lazily evaluated statics in Rust. https://github.com/rust-lang-nursery/lazy-static.rs

[34] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. 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.

[35] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Shanghai, China.

[36] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. Lincoln, NE.

[37] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. 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.

[38] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST*

'13). San Jose, CA.

[39] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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.

[40] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. 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.

[41] Brandon Lucia and Luis Ceze. 2009. 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.

[42] Ricardo Martins. 2016. Interior mutability in Rust, part 2: thread safety. https://ricardomartins.cc/2016/06/25/interior-mutability-thread-safety

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

[44] MSRC. 2019. Why Rust for safe systems programming. https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming

[45] Octoverse. 2019. The State of Octoverse. https://octoverse.github.com/

[46] Stack Overflow. 2016. Stack Overflow Developer Survey 2016. https://insights.stackoverflow.com/survey/2016#technology-most-loved-dreaded-and-wanted

[47] Stack Overflow. 2017. Stack Overflow Developer Survey 2017. https://insights.stackoverflow.com/survey/2017#most-loved-dreaded-and-wanted

[48] Stack Overflow. 2018. Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted

[49] Stack Overflow. 2019. Stack Overflow Developer Survey 2019. https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

[50] Alex Ozdemir. 2019. Unsafe in Rust: Syntactic Patterns. https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax

[51] Alex Ozdemir. 2019. Unsafe in Rust: The Abstraction Safety Contract and Public Escape. https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-escape

[52] Parity-ethereum. 2019. The Parity Ethereum Client. https://github.com/paritytech/parity-ethereum/pull/11172

[53] Parity-ethereum. 2019. The Parity Ethereum Client. https://github.com/paritytech/parity-ethereum/pull/11175

[54] Parity-ethereum. 2019. The Parity Ethereum Client. https://github.com/paritytech/parity-ethereum/issues/11176

[55] Quantum. 2019. Quantum. https://wiki.mozilla.org/Quantum

[56] Rand. 2019. Rand. A Rust library for random number generation. https://github.com/rust-random/rand

[57] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS '17)*. San Diego, CA, USA.

[58] Rayon. 2019. A data parallelism library for Rust. https://github.com/rayon-rs/rayon

[59] Redox. 2019. The Redox Operating System. https://www.redox-os.org/

[60] Redox. 2019. The Redox Operating System. https://gitlab.redox-os.org/redox-os/relibc/issues/159

[61] Eric Reed. 2015. *Patina: A Formalization of the Rust Programming Language.* Technical Report UW-CSE-15-03-02. University of Washington.

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

[63] Rust-book. 2019. Validating References with Lifetimes. https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

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

[65] Rust-nomicon. 2019. The Rustonomicon. https://doc.rust-lang.org/nomicon/

[66] RustSec. 2019. Security advisory database for Rust crates. https://github.com/RustSec/advisory-db

[67] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems, 15(4):391-411* (1997).

[68] Servo. 2019. The Servo Browser Engine. https://servo.org/

[69] Sid Shanker. 2018. Safe Concurrency with Rust. http://squidarth.com/rc/rust/2018/06/04/rust-concurrency.html

[70] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS '16)*. San Diego, CA, USA.

[71] Straits. 2019. Stratis: Easy to use local storage management for Linux. https://stratis-storage.github.io/

[72] Mingshen Sun, Yulong Zhang, and Tao Wei. 2018. When Memory-Safe Languages Become Unsafe. In *DEF CON China (DEF CON China '18')*. Beijing, China.

[73] Benchmarks Game Team. 2019. Rust versus C gcc fastest programs. https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust.html

[74] The Rust Team. 2019. Rust Empowering everyone to build reliable and efficient software. https://www.rust-lang.org/

[75] Threadpool. 2019. A very simple thread pool for parallel task execution. https://github.com/rust-threadpool/rust-threadpool

[76] Tikv. 2019. A distributed transactional key-value database. https://tikv.org/

[77] Tock. 2019. Tock Embedded Operating System. https://www.tockos.org/

[78] Tock. 2019. unnecessary unsafe tag. Tock issue #1298. https://github.com/tock/tock/issues/1298

[79] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Gothenburg, Sweden.

[80] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. 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.

[81] Aaron Turon. 2015. FeFearless Concurrency with Rust. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[82] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. 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.

[83] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. POMP: Postmortem Program Analysis with Hardware-enhanced Post-crash Artifacts. In *Proceedings of the 26th USENIX Conference on Security Symposium (Security '17)*. Vancouver, Canada.

[84] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, Xiaofeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *Proceedings*

of the 40th IEEE Symposiums on Security and Privacy (Oakland '19).

[85] Jie Yu and Satish Narayanasamy. 2009. 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.

[86] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. 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.

[87] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium (Usenix Security '18)*. Berkeley, CA, USA.

[88] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. 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.

[89] Wei Zhang, Chong Sun, and Shan Lu. 2010. 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.