Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs

HUI XU, School of Computer Science, Fudan University
ZHUANGBIN CHEN, Department of CSE, The Chinese University of Hong Kong
MINGSHEN SUN, Baidu Security
YANGFAN ZHOU, School of Computer Science, Fudan University and Shanghai Key Laboratory of Intelligent Information Processing
MICHAEL R. LYU, Department of CSE, The Chinese University of Hong Kong

3

Rust is an emerging programming language that aims at preventing memory-safety bugs without sacrificing much efficiency. The claimed property is very attractive to developers, and many projects start using the language. However, can Rust achieve the memory-safety promise? This article studies the question by surveying 186 real-world bug reports collected from several origins, which contain all existing Rust common vulnerability and exposures (CVEs) of memory-safety issues by 2020-12-31. We manually analyze each bug and extract their culprit patterns. Our analysis result shows that Rust can keep its promise that all memory-safety bugs require unsafe code, and many memory-safety bugs in our dataset are mild soundness issues that only leave a possibility to write memory-safety bugs without unsafe code. Furthermore, we summarize three typical categories of memory-safety bugs, including automatic memory reclaim, unsound function, and unsound generic or trait. While automatic memory claim bugs are related to the side effect of Rust newly-adopted ownership-based resource management scheme, unsound function reveals the essential challenge of Rust development for avoiding unsound code, and unsound generic or trait intensifies the risk of introducing unsoundness. Based on these findings, we propose two promising directions toward improving the security of Rust development, including several best practices of using specific APIs and methods to detect particular bugs involving unsafe code. Our work intends to raise more discussions regarding the memory-safety issues of Rust and facilitate the maturity of the language.

CCS Concepts: • **Software and its engineering → Language types**; **Software defect analysis**; • **Security and privacy → Software security engineering;**

Additional Key Words and Phrases: Rust, memory-safety bugs, common vulnerability and exposures

This work was supported by China Education and Research Network under Grant No. NGII20190410 and the Research Grants Council of the Hong Kong Special Administrative Region, China under Grant No. CUHK 14210717 of the General Research Fund.

Authors' addresses: H. Xu, School of Computer Science, Fudan University, Shanghai 200438, China; email: xuh@fudan.edu.cn; Z. Chen and M. R. Lyu, Department of CSE, The Chinese University of Hong Kong, Shatin, N.T., China; emails: {zbchen, lyu}@cse.cuhk.edu.hk; M. Sun, Baidu Security, Sunnyvale, CA 94089; email: sunmingshen@baidu.com; Y. Zhou (corresponding author), School of Computer Science, Fudan University, Shanghai 200433, China and Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 200433, China; email: zyf@fudan.edu.cn.

1049-331X/2021/09-ART3 $15.00
https://doi.org/10.1145/3466642

ACM Reference format:
Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 3 (September 2021), 25 pages.
https://doi.org/10.1145/3466642

1 INTRODUCTION

Memory-safety bugs (e.g., buffer overflow) are critical software reliability issues [35]. Such bugs commonly exist in software written by system programming languages like C/C++ that allow arbitrary memory access. The problem is challenging because regulating memory access while not sacrificing the usability and efficiency of the language is difficult. Rust is such a system programming language that aims at addressing the problem. Since the release of its stable version in 2015, the community of Rust grows very fast, and many popular projects are developed with Rust, such as the web browser Servo [3], and the operating system TockOS [24].

To achieve memory-safety programming, Rust introduces a set of strict semantic rules for writing compiling code and therefore preventing bugs. The core of these rules is an **ownership-based resource management** (**OBRM**) model, which introduces an owner to each value. Ownership can be borrowed among variables as references or aliases in two modes: mutable or immutable. The main principle is that a value cannot have multiple mutable aliases at one program point. Once a value is no longer owned by any variable, it would be dropped immediately and automatically. In order to be more flexible and extensible, Rust also supports unsafe code which may break the principle but should be denoted with an `unsafe` marker, meaning that developers should be responsible for the risk of using such code. Existing memory-safety checking mechanisms enforced by Rust compiler are unsound for unsafe code.

As Rust surges into popularity, a critical concern to the software community is how Rust performs in combating memory-safety bugs. Previous to our work, Evans et al. [15] have performed a large-scale study showing that unsafe code is widely used in Rust crates (projects). However, it remains unclear whether and how unsafe code could undermine the memory safety of real-world Rust programs. Since existing materials lack an in-depth understanding, we attempt to address this question by empirically investigating a set of critical bugs reported in real-world Rust projects. When we are preparing this work, we notice another independent work [30] also studies the same problem. While their work has developed some understandings similar to our article, our analysis involves more memory-safety bugs and provides several novel findings different from them, such as the culprit patterns of using generic or trait. We will clarify the differences in Section 8.1.

To elaborate, our work collects a dataset of memory-safety bugs from several origins, including Advisory-DB, Trophy Case, and several open-source projects on GitHub. The dataset contains 186 memory-safety bugs in total, and it is a superset of all Rust **common vulnerability and exposures** (**CVEs**) with memory-safety issues by 2020-12-31. For each bug, we manually analyze the consequence and culprit and then classify them into different groups. Since the consequences are nothing special in comparison with those of traditional C/C++ [35], we adopt a top-down approach and directly employ the labels like buffer overflow/over-read, use-after-free, and double free. However, we have no prior knowledge regarding the patterns of culprits in Rust, so we adopt a bottom-up approach and group them if two culprits demonstrate similar patterns.

Based on these bugs, this article studies three major questions. The first question is how effective is Rust in preventing memory-safety bugs. Our inspection result shows that all bugs in our dataset require unsafe code except one compiler bug. The main magic lies in that Rust does not tolerate

unsound APIs provided by either its standard library (std-lib) or third-party libraries. As a result, many memory-safety bugs are mild issues merely introducing unsound APIs, and they cannot accomplish memory-safety crimes by themselves. Therefore, we can conjecture that Rust has kept its promise that developers are unable to write memory-safety bugs without using unsafe code.

Our second question concerns the typical characteristics of memory-safety bugs in Rust. To this end, we have extracted three typical categories of memory-safety bugs, which are automatic memory reclaim, unsound function, and unsound generic or trait. Auto memory reclaim bugs are related to the side effects of Rust OBRM. Specifically, Rust compiler enforces automatic destruction of unused values, i.e., by calling `drop(v)` where `v` outlives its usefulness. The design is supposed to mitigate the memory leakage issue because Rust has no runtime garbage collection, but it leads to many use-after-free, double free, and freeing invalid pointer (due to uninitialized memory access) issues in our dataset. The default stack unwinding strategy of Rust further intensifies such side effects. The category of unsound function manifests the essential challenge for Rust developers to avoid unsoundness issues in their code. The category of unsound generic or bound reveals another key challenge faced by Rust developers for delivering sound APIs when employing sophisticated language features like polymorphism and inheritance. Besides, there are also some bugs that do not belong to the previous categories, and they are mainly caused by common mistakes, such as arithmetic overflow or boundary checking issues, which are not our interest.

Our third question studies the lessons toward making Rust programs more secure. On one hand, we find several common patterns of bugs and patches that can be employed as the best practices for code suggestion, such as bounding the generic-typed parameters with `Send` or `Sync` trait when implementing `Send` or `Sync` trait, or disabling automatic drop (with `ManuallyDrop<T>`) as earlier as possible. On the other hand, we may extend the current static analysis scheme of Rust compiler to support unsafe code so as to detect some specific types of bugs, especially those related to automatic memory reclaim.

We highlight the contribution of our article as follows.

- This article serves as a pilot empirical study of Rust in preventing memory-safety bugs. It performs an in-depth analysis regarding the culprits of real-world memory-safety bugs and generates several major categories, including automatic memory reclaim, unsound function, unsound generic or trait, and other errors.
- We highlight that a unique challenge of Rust development is to deliver sound APIs, and using advanced features of Rust (e.g., generic or trait) intensifies the risks of introducing unsoundness issues. Furthermore, we underline that automatic memory reclaim bugs are related to the side effect of Rust newly-adopted OBRM, and the default stack unwinding strategy of Rust further exacerbates such problems.
- Our work proposes two promising directions toward improving the security of Rust development. To developers, we provide several best practices for helping them write sound code. To compiler/tool developers, our suggestion is to consider extending the current safety checking to the realm of unsafe code, especially in detecting the bugs related to automatic memory reclaim.

The rest of this article is organized as follows. Section 2 reviews the memory-safety war of Rust; Section 3 introduces our study approach; Section 4 overviews the characteristics of our collected bugs; Section 5 demonstrates the detailed categories and patterns of these bugs. Section 6 discusses the lessons learned; Section 7 clarifies the threats of validity; Section 8 presents related work; finally, Section 9 concludes the article.

2 PRELIMINARY

This section reviews the preliminary of memory-safety bugs and discusses the mechanisms of Rust in preventing them.

2.1 Memory-Safety Bugs

Memory-safety bugs are serious issues for software systems. According to the statistical report of MITRE,[1] memory-safety bugs are enumerated among the top dangerous software vulnerabilities. Next, we review the concept of memory-safety bugs and discuss why preventing them is hard.

2.1.1 Concept of Memory-Safety Bugs. In general, memory-safety bugs are caused by arbitrary memory access. Below, we discuss several typical types of memory-safety issues.

- *Use-after-free*: After freeing a pointer, the buffer pointed by the pointer would be deallocated or recycled. However, the pointer still points to the deallocated memory address, known as a dangling pointer. Dereferencing a dangling pointer causes *use-after-free* issues. Such bugs are dangerous because the freed memory may have already been reallocated for other purposes, or it may allow users to control the linked list of free memory blocks maintained by the system (e,g., with ptmalloc or tcmalloc [18]) and achieve arbitrary write [38].
- *Double free*: Freeing a pointer twice causes *double free* issues. Similar to use-after-free on heap, double free leaves room for attackers to manipulate the linked list of free memory blocks. Consequently, attackers may overwrite any memory slot (e.g., global offset table) of the process leveraging memory allocation system calls [14].
- *Buffer overflow/over-read*: Accessing a memory slot beyond the allocated boundary is undefined. It could happen in two situations: an out-of-bound offset (e.g., due to boundary check error) or an in-bound offset with an invalid unit size (e.g., due to type conversion or memory alignment issue).
- *Uninitialized memory access*: It means the memory is used without proper initialization. Intuitively, it may leak the old content. If the memory contains values of pointer types, falsely accessing or freeing the pointer would cause *invalid memory access*.

Note that some *concurrency bugs* may also incur memory-safety issues due to data race or race conditions. The consequences could be use-after-free, buffer overflow, *and so on*. However, not all concurrency bugs involve memory-safety issues, such as deadlock,[2] or logical errors that simply crash the program without segmentation faults.[3] Therefore, this work does not treat concurrency issues as a sub-type of memory-safety bugs but another different bug type orthogonal to memory-safety issues.

2.1.2 Challenges in Preventing Memory-Safety Bugs. As long as developers use pointers, eradicating memory-safety bugs is very difficult if not impossible. Classical strategies to combat these bugs are mainly two-fold. One is to design memory-safety protection mechanisms, such as stack canary and shadow stacks [12] for protecting stack integrity, ASLR [16] against code reuse, glibc fasttop for detecting double free, *and so on*. Although such mechanisms are effective, they only raise the bar of attacks but cannot prevent them [19]. The other strategy is to prevent introducing memory-safety bugs from the beginning, such as using type-safe languages [33] and banning raw pointers, like Java. However, the enforcement of the memory-safety feature may also

[1] https://cwe.mitre.org/top25.
[2] https://github.com/diem/diem/pull/4479.
[3] https://github.com/diem/diem/pull/5190.

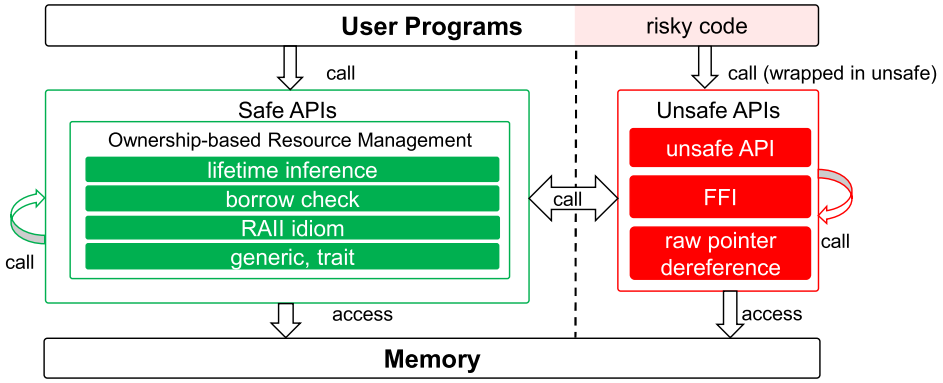


Fig. 1. Idea of Rust for preventing memory-safety bugs.

pose limitations to the language, making it inefficient for system-level software development with rigorous performance requirements.

2.2 Memory-Safety of Rust

Rust is a system programming language that aims at preventing memory-safety bugs while not sacrificing performance. It approaches the goal by introducing a set of strict semantic rules at the compiler level. In this way, Rust can be more efficient than other programming languages (e.g., Go) that rely much on runtime memory checking and garbage collection [10].

2.2.1 Basic Design. Figure 1 overviews the idea of Rust. Rust is in nature a hybrid programming language, including a safe part and an unsafe part. The safe part guarantees that the behaviors of all the code and APIs are well-defined, and programs using safe APIs only should have no risk of memory-safety issues. The unsafe part has no such guarantee and may lead to undefined behaviors, but it is necessary to meet some specific needs, e.g., low-level memory access for software development with rigorous performance requirements. Any code that may lead to undefined behaviors should be declared as unsafe, such as dereferencing raw pointers, calling **foreign function interfaces (FFIs)** or unsafe APIs. Actually, many safe APIs also employ unsafe APIs internally, and these APIs can be safe because they have got rid of all memory-safety risks, e.g., via conditional code. A function calling unsafe APIs can be declared as either safe or unsafe, which mainly depends on the developer's decision. Rust cannot check whether the declaration is correct or not. Therefore, falsely declaring an API as safe is dangerous and could undermine the soundness of safe Rust. Besides, any code that may cause memory leakage is safe in Rust because memory leakage does not directly lead to security problems but mainly causes performance degradation or availability issues [37]. Therefore, our following discussion will not include memory leakage issues.

The core of safe Rust is a novel ownership-based resource management model [21]. Ownership means a value should have one variable or identifier as its owner, and ownership is exclusive that a value can have only one owner. However, ownership can be borrowed among variables as references or aliases in two modes: immutable (default) or mutable (with an extra `mut` marker). The model assumes that only one variable can have mutable access to a value at any program point while other aliases have neither mutable nor immutable access at that point, or only immutable access can be shared among multiple aliases. Rust implements a *borrow checker* in its compiler to achieve the goal. The borrowed ownership expires automatically when the program execution exits the scope. If a value is no longer owned by variables, it would be dropped immediately to reclaim the buffer no longer used. Associated with the model, Rust introduces a *lifetime*

inference mechanism (similar as type inference [29]) which assures that the lifetime of a borrowed ownership would last long enough for use. Together, they form a basis for Rust in preventing memory-safety bugs. The realization of ownership-based resource management is based on the **resource acquisition is initialization (RAII)** idiom. RAII emphasizes resource allocation during object creation with the constructor and resource deallocation during object destruction with the destructor [31, 34]. Therefore, the borrow-check and lifetime-inference algorithms only need to consider well-initialized objects and their references. In other words, OBRM saves Rust compiler from solving complex pointer analysis problems and thus largely simplifies the game played by the compiler.

The OBRM model applies to all types supported by Rust, including self-defined structs, generics, and traits. Generic relates to the polymorphism feature of Rust, which means an unspecified type (often denoted with `T`) that will be determined during compilation (similar to C++ template). Trait is similar to the interface of Java yet also relates to the inheritance feature of Rust, which means a trait type can be inherited and its member functions can be reimplemented by the derived types, such as the `Clone` trait and `Drop` trait. When implementing Rust code, a generic can be bounded with some traits to be sound.

2.2.2 Principle for Preventing Memory-Safety Bugs. OBRM lays the foundation of Rust in preventing memory-safety bugs. Via OBRM and other related design, Rust compiler assures Rust developers that they will not suffer memory-safety issues if not using unsafe code. Below, we justify the effectiveness of OBRM in preventing different memory-safety issues.

— *Preventing dangling pointer*: According to the OBRM, variables or pointers should be initialized with values when defining them. This enables the compiler to trace the abstract states (ownership and lifetime) of the value and perform sound program analysis. Such analysis guarantees that safe Rust can prohibit shared mutable aliases, and therefore gets rid of the risks of dereferencing or freeing pointers whose values have been freed with another alias. Note that while defining raw pointers is valid in safe Rust, dereferencing them is only allowed as unsafe. However, the soundness of Rust compiler would be invalidated when using unsafe code. For example, unsafe code could introduce shared mutable aliases and therefore makes the program vulnerable to double free due to the automatic memory reclaim scheme. We will elaborate more on this problem in Section 5.1.

— *Preventing buffer overflow/over-read*: OBRM also benefits in-range buffer access because each variable or pointer points to a value of a particular type, such as `i32`. It guarantees that memory data are properly aligned. For advanced data containers of Rust std-lib, such as `Vec<T>`, Rust generally maintains a length field for the object and performs boundary checks automatically during runtime. Again, these mechanisms are only sound for safe Rust, unsafe code (such as unsafe type conversion or dereferencing raw pointers) may lead to out-of-range access.

— *Preventing uninitialized memory access*: Safe Rust does not allow uninitialized memory by default. For example, creating a buffer with `uninitialized()` or `alloc()` is unsafe; reserving the capacity for a vector is safe, but directly increasing its length with `set_len()` is unsafe.

OBRM also applies to concurrent programming and therefore can mitigate memory-safety risks incurred by data race. For example, when accessing a variable defined in the main thread from a child thread, a marker `move` is needed to transfer the ownership of the variable. To access variables concurrently in multiple processes, Rust provides `Mutex<T>` and `Arc<T>` that achieve mutual exclusive lock and atomic access, respectively, and the type system of Rust ensures that

ownerships cannot be shared among threads unless they implementing Send or Sync traits, e.g., based on Mutex<T> or Arc<T>. These features can help developers avoid introducing data race or race conditions in their code.

Another essential trick of Rust is that safe Rust does not tolerate unsoundness, i.e., if a code snippet provides some safe APIs but which can be employed by developers to accomplish undefined behaviors without using unsafe code, the code snippet contains unsoundness. Rust requires that all safe APIs should not be able to incur undefined behaviors, otherwise it should be unsafe. This requirement applies to all Rust projects and developers, and it plays an important role as the Rust community gradually grows and many third-party libraries become available. Only when all these libraries are sound, the soundness of Rust compiler can be guaranteed.

3 OUR APPROACH

3.1 Objective

This work aims at reviewing the memory-safety game played by Rust. In particular, we are interested in three major research questions.

RQ-1. How effective is Rust in preventing memory-safety bugs? Rust promises developers that writing programs without using unsafe code should have no memory-safety risks. Therefore, we will study whether unsafe code is necessary for accomplishing real-world memory-safety bugs and how bad are these residual bugs. Answering these questions provides an overall assessment of Rust in combating memory-safety issues, which is especially useful to its potential users.

RQ-2. What are the characteristics of memory-safety bugs in Rust? For those residual memory-safety bugs in real-world Rust programs, what are their causal culprits? Are there any unique patterns in comparison with C/C++ bugs? Interpreting such issues can provide more insights toward understanding the problem better or mitigating them in the future.

RQ-3. What lessons can we learn in order to make Rust more secure? Based on the findings of RQ1 and RQ2, we shall be able to derive some useful knowledge, helping Rust developers further mitigate residual memory-safety issues. For example, we can extract some common code patterns that are risky to memory-safety issues and warn developers to use them carefully. Likewise, the characteristics of bugs may provide essential information for compiler developers to consider improving the language toward more resilience.

3.2 Data Collection

The first step of our study is to collect real-world memory-safety bugs. Thanks to GitHub, many influential Rust projects are maintained online and allow us to track the bugs and patches for particular projects. However, since GitHub is free-organized, the formats and qualities of bug reports vary a lot. Such limitations pose difficulties for automatically extracting and analyzing bugs in-depth. As our research does not focus on proposing automatic analysis approaches, we prefer manual inspection, which can be performed in an in-depth and precise manner. While manual inspection can be in-depth, it sacrifices scalability. Therefore, we try to find the most representative cases for manual inspection.

We collect bugs from four origins, Advisory-DB, Trophy Cases, official Rust compiler project, and several other Rust projects on GitHub.

— *Advisory-DB*[4] is a repository that collects security vulnerabilities found in Rust software, and it is a superset of existing CVEs[5] related to Rust programs. The dataset contains over

[4]https://github.com/RustSec/advisory-db.
[5]https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust.

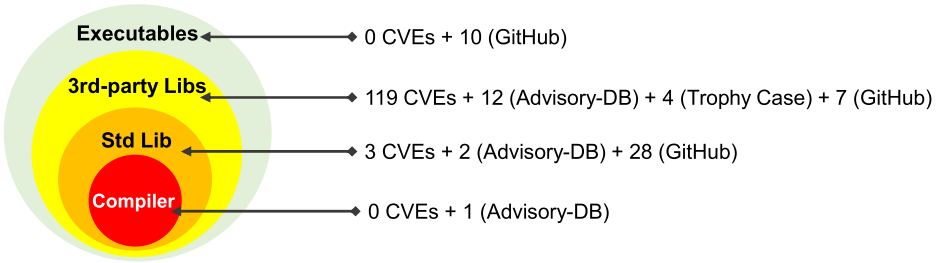


Fig. 2. Dataset of memory-safety bugs in our study.

200 entries from 100+ contributors by 2020-12-31. However, not all entries are memory-safety bugs, such as those labeled with unmaintained, crypto-failure, or denial-of-service. We finally obtained 137 memory-safety bugs (including 122 CVEs) for further study. Note that the bug selection process is via manual inspection because the original entries of Advisory-DB are provided by different developers, and they may employ different labels.

— *Trophy Case*[6] collects a list of software bugs found by fuzzing tools. The bug reports originate from the users of fuzzing tools for Rust programs, such as cargo-fuzz.[7] The repository contains over 200 bug entries from 30+ contributors by 2020-12-31. Unfortunately, most of these bugs do not involve memory-safety issues, and only six are labeled with a security tag, including two duplicated ones with CVE-IDs.

— *Rust Compiler Project* maintained on GitHub also contains some bugs that may lead to undefined behaviors. In particular, we are interested in the unsoundness bugs introduced in Rust standard libraries. To this end, we search GitHub issues with the label "I-unsound" and "T-libs" or "impl-libs" and inspect each of them. Note that we mainly keep closed issues for further study because they can provide us thorough information about the bug and its fix. We finally obtained 33 memory-safety bugs in Rust std-lib, including five duplicated ones with Advisory-DB (three have CVE-IDs).

— *Other Projects.* To further extend our dataset, we choose five highly rated (over 500 stars) Rust projects on GitHub. We search the keywords related to memory-safety (e.g., "core dumped", "segmentation fault", "use-after-free") for finding the historical bugs occurred in these repositories. We finally obtained 18 bugs in these projects, including one duplicated CVE. Note that since the bugs of previous origins are mainly library bugs, we especially choose three projects of executables.

We finally get 186 memory-safety bugs for further study. Figure 2 provides an overview of these bugs according to the types of their host programs, one bug in Rust compiler, 33 in Rust standard library, 142 in third-party libraries, and 10 in executable programs. A full list of these bugs and their detailed information are available online.[8]

Since Rust does not consider memory leakage as memory-safety bugs, we exclude these bugs from our dataset. In particular, many CVEs with memory leakage issues lead to Denial of Service but not memory-safety problems. For example, a malformed input or unexpected parameter value could cause the system to preallocate a huge memory space, such as CVE-2018-20993 and CVE-2019-15544. Besides, employing recursive function calls may overflow the stack (e.g., 8MB on Linux X86_64 by default) if the recursion is too deep, such as CVE-2019-2500. Such bugs mainly lead to

[6] https://github.com/rust-fuzz/trophy-case.
[7] https://github.com/rust-fuzz/cargo-fuzz.
[8] https://github.com/Artisan-Lab/Rust-memory-safety-bugs.

DoS because the operating system can detect such issues and kill the process. Besides, some other memory bugs might also be confused with memory-safety ones, but they do not incur security issues. Typically, a bug with "index out-of-bound" error indicates the violation of index validity check performed by the container (e.g., a vector) itself during runtime. There are many such cases founded by fuzzing tools, like those reported by Trophy Cases.[9] Similar to memory leakage, these bugs can affect the usability of the program but have no security risks. We will not study these bugs in our article.

3.3 Bug Analysis Method

We are interested in the consequences and culprits of each memory-safety bug. For the consequences, we adopt a top-down approach. Because the taxonomy of memory-safety bugs has been well studied in the literature (e.g., [35]), we simply adopt the existing categorization method, such as buffer overflow/over-read, use-after-free, double free, and uninitialized memory access. For the culprits, we employ a *bottom-up approach*, i.e., we cluster similar bugs while do not assume any prior categories. While the bug description can provide us some hints, we also manually check the commit of each bug fix on GitHub and analyze their culprits. The analysis is mainly done by the first author and checked by the second and third authors. If there are conflicts or uncertainties in categorizing a bug, we review the bug description together to determine the most appropriate label for the bug. By comparing the buggy code and bug fixes, we can locate the root cause of each bug and confirm the culprits. In particular, many bugs, especially most CVEs employ similar descriptions, which provide a basis to form our culprit categories. For example, the description of CVE-2020-36206 "*lack of Send and Sync bounds*" is similar to CVE-2020-36220 "*Demuxer<T> omits a required T:Send bound*", and we cluster them into one group with a label "*insufficient bound of generic*"; the description of CVE-2020-36210 "*uninitialized memory can be dropped when a panic occurs*" is similar to CVE-2020-25795 "*insert_from can have a memory-safety issue upon a panic*" and we cluster them into another group with a label "*bad drop at cleanup block*". For bugs without clear descriptions, we will first try to categorize them into existing groups or employ a new group if their patterns are different. In this way, we finally obtain four major categories and 13 sub-categories. The details will be discussed in the next section (Section 4.1).

4 OVERVIEW OF MEMORY-SAFETY BUGS

This section provides an overview of our bug analysis results and then addresses RQ-1.

4.1 Distribution of Bugs

We categorize memory-safety bugs based on their consequences and culprits. Table 1 overviews the statistics of our analysis result. The table header lists five major consequences we adopted in this article, buffer overflow/over-read, use-after-free, double free, uninitialized memory access, and other undefined behaviors (i.e., memory-safety bugs that do not directly lead to the previous four consequences). We cluster the culprits of bugs into four major categories and 13 sub-categories. Each cell of the table presents the number of corresponding bugs with a particular culprit and consequence. If a bug has several different culprits or can lead to different consequences based on their usage, we only count the major culprit and consequence. For example, many use-after-free bugs may also lead to double free, but we mainly categorize them into use-after-free. To facilitate our following discussion, we also present the fine-grained distribution of bugs based on their origins (i.e., std-lib, CVEs, or others sources), separated with "+".

[9] https://github.com/rust-fuzz/trophy-case.

Table 1. Distribution of Memory-Safety Bugs in Rust Std-lib + CVEs + Others

Culprit		Consequence					Total
		Buf. Over-R/W	Use-After-Free	Double Free	Uninit Mem	Other UB	
Auto Memory Reclaim	Bad Drop at Normal Block	0 + 0 + 0	1 + 9 + 6	0 + 2 + 1	0 + 2 + 0	0 + 1 + 0	22
	Bad Drop at Cleanup Block	0 + 0 + 0	0 + 0 + 0	1 + 7 + 0	0 + 5 + 0	0 + 0 + 0	13
Unsound Function	Bad Func. Signature	0 + 2 + 0	1 + 5 + 2	0 + 0 + 0	0 + 0 + 0	1 + 2 + 4	17
	Unsoundness by FFI	0 + 2 + 0	5 + 1 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	12
Unsound Generic or Trait	Insuff. Bound of Generic	0 + 0 + 1	0 + 33 + 2	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	36
	Generic Vul. to Spec. Type	3 + 0 + 1	1 + 0 + 0	0 + 0 + 0	1 + 0 + 1	1 + 2 + 0	10
	Unsound Trait	1 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 2 + 0	6
Other Errors	Arithmetic Overflow	3 + 1 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	5
	Boundary Check	1 + 9 + 0	1 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 0 + 0	12
	No Spec. Case Handling	2 + 2 + 1	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	2 + 1 + 1	9
	Exception Handling Issue	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	0 + 0 + 0	1 + 2 + 1	4
	Wrong API/Args Usage	0 + 3 + 0	1 + 4 + 0	0 + 0 + 0	0 + 1 + 1	0 + 5 + 2	17
	Other Logical Errors	0 + 4 + 1	2 + 3 + 4	0 + 0 + 1	0 + 1 + 0	1 + 4 + 1	22
Total		40	82	12	12	39	185

For Simplicity, We Count the CVEs of Rust Std-lib into Rust Std-lib.

Next, we present the high-level concepts of the four major culprit categories. While the first three ones are more or less typical for Rust, other errors are mainly common mistakes not interest to us.

— *Automatic Memory Reclaim*: Bugs of this group imply memory-safety issues related to the automatic memory reclaim mechanism of OBRM. According to the place where the culprit drop statement happens, we divide this group into two sub-groups, *bad drop at normal block* and *bad drop at cleanup block*.
— *Unsound Function*: This group includes two sub-groups, *bad function signature,* which means developers publish an API with an incorrect function signature making the API unsound, and *unsoundness by FFI,* which means developers call an FFI but do not handle their behaviors properly.
— *Unsound Generic or Trait*: Generic and trait are advanced features of Rust type system to support polymorphism and inheritance. Bugs of this group relate to the soundness issues of generics and traits, including *insufficient bound of generic*, *generic vulnerable to specific type*, and *unsound trait*.
— *Other Errors*: This group contains bugs that cannot be grouped into the previous three groups. These bugs are mainly due to various common mistakes, such as arithmetic overflow, boundary check, no special case handling, exception handling issue, wrong API usage, and other logical errors. When combining with unsafe code, these mistakes can also lead to memory-safety issues.

From Table 1, we can observe that use-after-free (82 bugs) is the most popular issue, and their major culprits include insufficient bound of generic (35 bugs), automatic memory reclaim (16 bugs), bad function signature (8 bugs), and so on. Besides, there are 40 buffer overflow/over-read, 12 double free bugs, 12 uninitialized memory access, and 39 bugs with other undefined behaviors. Due to the automatic memory reclaim scheme, most use-after-free bugs in Rust may also lead to double free issues because the dangling pointer is likely to be freed automatically after use. We will discuss the issue later in Section 5.

Among all the 185 bugs (excluding one compiler bug), 35 bugs have culprits directly related to the automatic memory reclaim issue, 52 related to unsound generic or trait, 29 related to unsound functions, and 69 are due to other errors. While bugs of the first three categories are of our interest because they are special in Rust, we are not interested in other errors because their patterns commonly exist in other languages. For example, arithmetic overflow and incorrect boundary

check are common mistakes, and the only difference for Rust is that these bugs involve unsafe read/write to trigger memory-safety problems.

4.2 Effectiveness of Rust

Based on the dataset, now we answer RQ-1. To this end, we first present our main result and then discuss the detailed analysis process by answering three sub-questions.

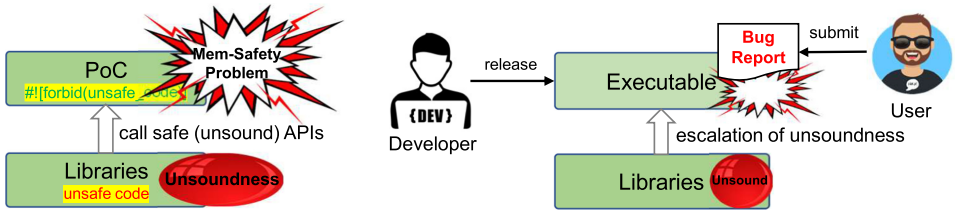
Result Highlight for RQ-1: All memory-safety bugs of Rust except compiler bugs require unsafe code. The main magic lies in that Rust enforces the soundness requirement on APIs provided by both Rust standard library or third-party libraries. As a result, many bugs in our dataset are mild soundness issues and these bugs have not incurred severe memory-safety problems. Besides, the trend of compiler bugs is much stable. Therefore, we can conclude the effectiveness of Rust in preventing memory-safety bugs.

RQ-1.1 Do all memory-safety bugs require unsafe code? According to the memory-safety promise of Rust, developers should not be able to write Rust code that contains memory-safety bugs if not using unsafe Rust. Therefore, we first examine if all these bugs require unsafe APIs. Our result confirms that all memory-safety bugs in our dataset except the compiler bug require unsafe code. The result is consistent with another work by Qin. et al. [30]. According to Figure 2, a bug may propagate from an inner circle to outer circles, i.e., compiler -> std-lib -> third-party lib -> executable. For example, an unsoundness issue of the Rust compiler generally implies accepting ill-formed source code or generating incorrect compiled code, and it may affect any programs being compiled, including std-lib. Our dataset contains only one compiler bug that accepts source code violating the lifetime rule of Rust. Consequently, developers may safely extend a bounded-lifetime value to static, leading to use-after-free bugs in the compiled code. This bug is the only compiler bug collected by Advisory-DB but not the sole one ever introduced to Rust compiler. For other memory-safety issues, they could be traced back to either its own unsafe code or other unsafe code of its required libraries. For example, a memory-safety issue of an executable program that forbids unsafe code could be caused by the unsoundness of std-lib.

As there are still many memory-safety bugs in Rust projects, we are interested in how severe are these bugs, especially the CVEs. This motivates our RQ-1.2. Besides, from the view of Rust users, they may treat Rust compiler and std-lib as black box. Therefore, the robustness of Rust compiler plays an essential role in justifying the effectiveness of Rust and motivates RQ-1.3.

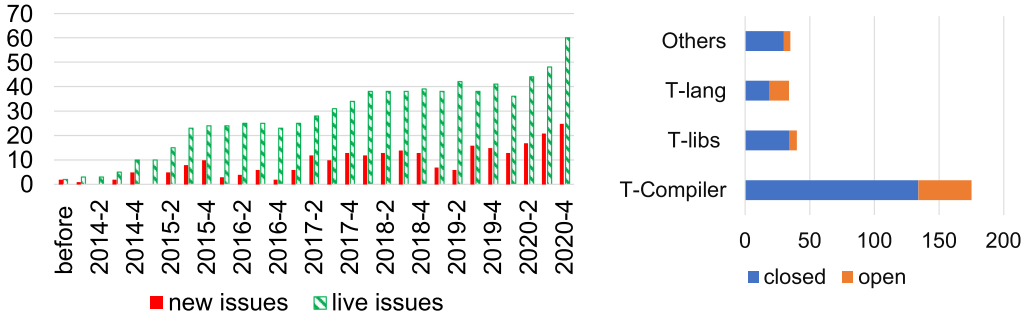
RQ-1.2 How severe are these memory-safety bugs? According to Figure 2, all existing CVEs are library bugs. This implies such bugs do not cause bad consequences directly unless the buggy APIs are employed by other executables. In comparison, many existing C/C++ CVEs are executable bugs [17]. Furthermore, an important phenomenon of these bugs is that many of them do not contain errors inside but merely introduce unsoundness that violates the memory-safety design of Rust. Consequently, they leave possibilities for their users to write memory-safety bugs without any unsafe marker. For example, all bugs with the culprit of insufficient bound of generic or bad function signature are API declaration issues. When using these APIs carefully, developers will not suffer memory-safety problems. However, since these bugs leave a hole for introducing memory-safety issues without unsafe code, they are also deemed as severe issues and have been assigned CVE-IDs. To demonstrate the existence of such unsoundness issues, bug reporters often need to write complicated **proof-of-concept** (**PoC**) samples attacking the vulnerability of unsound APIs.

Figure 3(a) demonstrates the attack model of Rust. It assumes users should trust all safe APIs provided by libraries that they will not cause memory-safety problems in any circumstances. However, via unsound APIs of libraries, attackers can breach the protection and write memory-safety bugs or PoCs even without unsafe code. Since the consequence and severity of unsound



(a) Attack model of Rust. (b) Escalation of unsound APIs to real-world bugs.

Fig. 3. The influence of Rust API unsoundness.



(a) Trend of unsoundness issues. (b) Classes of unsoundess issues.

Fig. 4. Unsoundness issues of the Rust compiler project on GitHub.

APIs depend on how they are employed, we have further investigated whether these bugs have been propagated to other programs based on their issues raised on GitHub. Figure 3(b) presents a framework demonstrating how unsound APIs escalate to real-world memory-safety problems. Our investigation result finds no such obvious escalations with any CVEs. The only propagated memory-safety bug is one in our randomly selected program `curl-rust`.[10]

Besides, we also find several other interesting phenomena. Firstly, many bugs are reported by security experts when auditing these Rust projects. For example, `Qwaz` (GitHub ID) has contributed 29 bugs in our dataset, and `ammaraskar` has contributed 18 bugs. Secondly, there are several similar bugs due to code clones, such as CVE-2020-36212 copies issue-60977 of std-lib, CVE-2020-36213 copies issue-78498 of std-lib, and issue-21186 of servo copies issue-51780 of std-lib. Thirdly, most project owners are serious about the unsoundness issues within their code while others may not. Taking CVE-2020-35876 as an example, after an unsoundness issue was raised, the project owner closed the issue without a patch but left a few words, "*Issues like this kill any motivation to improve thing. Not helpful. Open a PR instead*".

In short, our analysis reveals that many memory-safety bugs in our dataset are mild, and they only incur possibilities of introducing memory-safety issues without unsafe code. However, since Rust promotes memory-safety, the community is more rigid toward such risks.

RQ-1.3 How robust is Rust compiler? To answer this question, we have surveyed the unsoundness issues of the Rust compiler project on GitHub. Figure 4(a) presents the number of unsoundness issues raised from 2014 to 2020 in a quarterly manner. Overall, there are over two hundred unsoundness issues (labeled as "I-unsound") raised until 2020-12-31, which could violate the memory-safety promise of safe Rust. Most of these issues are also labeled with "T-Compiler",

[10] https://github.com/alexcrichton/curl-rust/issues/333.

which means assigned to the compiler team or compiler bugs. Besides, there are 40 issues labeled with "T-libs" (including "impl-libs") or "T-lang", which means assigned to the library team or language team, respectively. We can observe that although the trend of new issues does not decline, it is much stable. Why unsoundness issues can hardly be prevented? As the features of Rust keep getting enriched, introducing new unsoundness issues are unavoidable. Besides, some unsoundness issues related to the design of the language are difficult to fix. Taking CVE-2019-12083 as an example, the temporary fix is to destabilize the function, which raises a warning when employing it. However, this fix is imperfect, and the issue still keeps open since raised months ago.

5 TYPICAL PATTERNS OF MEMORY-SAFETY BUGS

This section studies the typical patterns of memory-safety bugs and answers RQ-2. Below, we first highlight our main result and then discuss these detailed culprit patterns.

Result Highlight for RQ-2: We find three major categories of memory-safety bugs typical for Rust: (1) automatic memory reclaim that uncovers the side effect of Rust OBRM which is originally proposed to prevent memory leakage based on static analysis, (2) unsound function that reveals the essential challenge for Rust developers to avoid unsoundness in their code, and (3) unsound generic or trait that unveils another advanced challenge for Rust developers to deliver sound APIs when employing sophisticated language features like polymorphism and inheritance.

5.1 Automatic Memory Reclaim

In order to implement OBRM and avoid memory leakage, Rust adopts RAII, i.e., the memory will be automatically acquired while initialization and released if it is no longer used [31]. Rust achieves automatic memory reclaim by adding `drop()` function calls for particular variables in **Mid-level Intermediate Representation (MIR)** code during compilation. However, since Rust compiler is unsound for unsafe code, such automatic memory reclaim scheme makes Rust programs prone to dangling pointers. In our dataset, 35 bugs are related to such side effect, including 23 bugs occurred during normal program execution and 12 during stack unwinding while an exception is triggered.

5.1.1 Bad Drop at Normal Execution Block . In general, automatic memory reclaim can only trigger memory-safety problems if there is unsafe code, which incurs inconsistencies with Rust OBRM and breaches the soundness of Rust compiler. Below, we discuss several representative scenarios.

Code 1. PoC of use-after-free and double free bugs due to automatic memory reclaim.

```
fn genvec() -> Vec<u8> {
    let mut s = String::from("a_tmp_string");
    /*fix2: let mut s = ManuallyDrop::new(String::from("a tmp string"));*/
    let ptr = s.as_mut_ptr();
    unsafe {
        let v = Vec::from_raw_parts(ptr, s.len(), s.len());
        /*fix1: mem::forget(s);*/
        return v;
        /*s is freed when the function returns*/
    }
}
fn main() {
    let v = genvec();
    assert_eq!('a' as u8,v[0]); /*use-after-free*/
    /*double free: v is released when the function returns*/
}
```

Code (2) C++ code: only x is dropped automatically when the function returns.

```cpp
struct ToDrop {
    string name;
    ToDrop(string name):name(name) {}
    ~ToDrop() {cout<<"Drop␣"<<name<<'\n';}
};

int main() {
    ToDrop x("x");
    ToDrop* y = new ToDrop("y");
    //delete y;
}
```

Code (3) Rust code: both x and y are dropped automatically when the function returns.

```rust
struct ToDrop { name:&'static st }
impl Drop for ToDrop {
    fn drop(&mut self) {
        println!("Drop␣{}", self.name);
    }
}
fn main() {
    let mut x = ToDrop { name:"x" };
    let y: *mut ToDrop
            = &mut ToDrop { name:"y" };
}
```

Fig. 5. Comparing the difference of C++ and Rust in automatic drop.

Typically, automatic memory reclaim would trigger memory-safety problems if the program violates the OBRM model with shared mutable aliases. As a result, dropping one alias would incur dangling pointers with the other one, which may further lead to use-after-free or double free. Code 1 demonstrates such a self-contained PoC based on CVE-2019-16140. The `genvec()` function composes a vector using an unsafe API `from_raw_parts()` and returns the vector. The unsafe API simply dereferences the pointer `ptr` and returns a reference, and it leaves the lifetime checking responsibility to developers. Unfortunately in this PoC, the pointer points to a temporary string `s` that would be destructed automatically when the function returns. Using the returned vector `v` in the main function would cause use-after-free issues. Dropping `v` directly would cause double free. To fix the bug, a common practice is to employ either `ManuallyDrop<T>` (e.g., by CVE-2018-20996) or `forget()` (e.g., by CVE-2019-16144) that can prevent calling the destructor of `T` when it expires.

In another similar case, shared aliases may not co-exist at the same program point. Instead, the pointer might be used later after the value is dropped, i.e., via usafe APIs or FFIs. CVE-2018-20997 and CVE-2020-35873 are examples of such cases. Note that the lifetime inference scheme associated with OBRM only takes references into consideration but not pointers.

As we have discussed, the problem occurs generally because the constructed variable does not exclusively own its value. Such issues are also popular when designing struct with raw pointers. If the struct does not own the memory pointed by the raw pointer after construction, the memory might be dropped earlier than the struct, e.g., CVE-2020-35711 and CVE-2020-35885. A useful patch to such bug is employing `PhantomData<T>`, which fools the compiler that it owns the data of type `T`. On the contrary, if a struct falsely turns a raw pointer into a reference, it may drop the memory it does not own, e.g., CVE-2020-35885.

The consequences of automatic memory reclaim are generally dangling pointers. However, if the memory has not been fully initialized, it may drop uninitialized memory (e.g., CVE-2020-35888), or incur other undefined behaviors related to the value being dropped (e.g., CVE-2020-35868).

Note that RAII is not a new concept proposed in Rust. For example, C++ also employs RAII, but its automatic destruction scheme is not as potent as Rust. For comparison, we demonstrate a toy example in Figure 5. Both the two code snippets create an object named `x` and a pointer that points to another object named `y`. When the function returns, both `x` and `y` are dropped automatically in Rust. However, C++ only drops `x` automatically, and developers need to delete `y` manually or use smart pointers instead.

5.1.2 Bad Drop at Cleanup Block. While the previous type of bugs relates to the side effects of automatic memory reclaim during normal execution, there are also several such bugs only occur

during stack unwinding. Rust compiler has two modes of runtime error handling: panic or abort. Panic is the default mode that performs stack unwinding automatically and reclaims the allocated resources; abort simply ends the process directly. This type of bugs only exists in the panic mode.

Memory-safety bugs that occurred during stack unwinding generally imply the existence of program points vulnerable to the cleanup routine, i.e., some memory operations have not been done to achieve a memory-safe state. If the program panics at these points, the cleanup routine may drop dangling or invalid pointers. Code 4 present a PoC that would access uninitialized memory if the program panics at line 5 and consequently drops an invalid pointer. Specifically, the function `read_from()` creates an object `foo` with `uninitialized::<Foo>()` and then initializes it with `src`. Therefore, the program points after creating `foo` are vulnerable until `foo` has been fully initialized. If the program panics at these vulnerable points, the stack unwinding process would call the destructor of `Foo`, which contains an invalid pointer in `Vec`. Examples of such bugs include CVE-2020-25794 and CVE-2020-36210. The underlying issue is that program points with uninitialized memory are themselves unsound because they violate the principle of RAII. Therefore, Rust introduces a new API `MaybeUninit` to meet such requirement and has deprecated `uninitialized()` recently. However, there are also other APIs that may also incur similar issues, such as `alloc()`.[11] Besides creating new objects, similar problems also exist when expanding a buffer. For example, CVE-2019-16138 expends a vector by firstly increasing its length with `set_len()` and then initializing the newly expended elements. In this way, the program points between those two steps are vulnerable to dropping uninitialized memory.

Code 4. PoC of dropping uninitialized memory during stack unwinding.

```
1  struct Foo { vec : Vec<i32>, }
2  impl Foo {
3     pub unsafe fn read_from(src: &mut Read) -> Foo {
4        let mut foo = mem::uninitialized::<Foo>();
5        //panic!(); /*panic here would recalim the uninitialized memory of type <Foo>*/
6        let s = slice::from_raw_parts_mut(&mut foo as *mut _ as *mut u8, mem::size_of::<Foo>());
7        src.read_exact(s);
8        foo
9     }
10 }
11 fn main() {
12    let mut v = vec![0,1,2,3,4,5,6];
13    let (p, len, cap) = v.into_raw_parts();
14    let mut u = [p as u64, len as _, cap as _];
15    let bp:*const u8 = &u[0] as *const u64 as *const _;
16    let mut b:&[u8] = unsafe { slice::from_raw_parts(bp, mem::size_of::<u64>()*3) };
17    let mut foo = unsafe{Foo::read_from(&mut b as _)};
18    println!("foo = {:?}", foo.vec);
19 }
```

If the vulnerable program points have shared mutable pointers, panicking at these points would incur double free issues. Note that in Code 1, one possible bug fix employs `forget(s)` to avoid dropping the temporary string `s`. However, if there are more lines of code that may panic the program after executing `from_raw_parts()` but before reaching `forget()`, these program points would contain shared mutable pointers to the memory of string `s`. Panicking at these points would incur double free. Examples of such bugs include CVE-2019-16880 and CVE-2019-16881.

[11] https://gitlab.com/dvshapkin/alg-ds/-/issues/1.

A common fix is to replace `forget()` with `ManuallyDrop<T>`. Actually, `forget()` is a wrapper of `ManuallyDrop<T>` but does not return a `ManuallyDrop<T>` object. Since `forget()` borrows the ownership of its parameter value and never returns, it is generally employed as late as possible so that other code can mutate the value. As a result, using `forget()` is prone to introducing program points vulnerable to stack unwinding, leading to double free or uninitialized memory access (e.g., CVE-2019-15552 and CVE-2019-15553). Besides employing `forget()` too late, similar problems also arise when shrinking a buffer with pointer elements. For example, if shrinking a vector by firstly copying the elements out and then calling `set_len()` to decrease its length, there would be vulnerable program points with duplicated data between the two operations. Such examples include CVE-2018-20991 and CVE-2020-25574.

5.2 Unsound Function

An unsound function could be attacked by users and lead to undefined behaviors without unsafe code. This section discusses the issue from two perspectives, bad function signature, and unsoundness introduced by FFIs. Note this category excludes various implementation issues of functions that may also lead to undefined behaviors, as well as wrong API usages that can be fixed easily.

5.2.1 Bad Function Signature. According to the composition of a function signature, developers may make mistakes in the following aspects.

- *Function safety*: If developers falsely declare an unsafe function as safe, users of the function may trigger undefined behaviors without using unsafe code themselves. In our dataset, most cases of bad function signature belong to the type, such as CVE-2019-15547, CVE-2019-15548, and CVE-2020-25016.
- *Parameter mutability*: If developers falsely define an immutable parameter but mutate it internally (e.g., by calling an FFI), users of the function may falsely think the parameter value is not changed and suffer undefined behaviors. Examples of such cases include CVE-2019-16142 and *so on*.
- *Lifetime bound*: Parameters of functions should be correctly bounded with lifetimes. Missing or insufficient lifetime bound might cause dangling pointers when using these functions, e.g., CVE-2020-35867 and CVE-2020-35879.
- *Function visibility*: In an extreme case, the whole function is unsound (e.g., deprecated or private) and should not be exposed to users, e.g., CVE-2020-35908.

5.2.2 Unsoundness by FFI. FFIs are only allowed in unsafe Rust because they could make Rust compiler unsound. To meet the security requirement of Rust, developers should make sure all the behaviors of FFIs are properly handled or declaring their caller functions as unsafe. Otherwise, the unsoundness of FFIs would propagate to safe Rust code and incur undefined behaviors.

In our dataset, most cases are due to the undefined behaviors of FFIs with particular applications. For example, issues-17207 of std-lib is unsound because the behavior of `mallocx()` in jemalloc is undefined for parameter size zero; issue-33770 of std-lib is unsound because the behavior of recursive lock is undefined with glibc. All these undefined behaviors are not properly handled. Moreover, FFIs interacting with the system may suffer race conditions, such as issue-27970 of std-lib and CVE-2020-26235, which are related to `getenv()` and `setenv()` of libc. Besides, there could be memory misalignment issues (e.g., CVE-2018-20998 and CVE-2020-35872) or inconsistent data layout (e.g., CVE-2020-35920 and CVE-2020-35921) when interacting with FFIs. Since such bugs are straightforward, we would not demonstrate their trivial details.

5.3 Unsound Generic and Trait

Generic and trait are advanced features of Rust type system to support polymorphism and inheritance, respectively. While such features can facilitate the development process, it also intensifies the risks of introducing unsoundness. Note that this category only considers the soundness issues of using the generic or trait but excludes various implementation issues of the traits or functions themselves.

5.3.1 Insufficient Bound of Generic. In Rust, a function or a struct can take a generic-type parameter as input, e.g., denoted as `T`, and the specific type of `T` will be determined during compilation based on how the function is called or how the struct is constructed. However, a practical function may not be able to support all types of variables with any lifetime, and hence bounds are required to make such restrictions. If the specified bounds are insufficient, it may incur memory-safe issues by passing parameters of unsupported types.

Code 5. PoC of lacking Send trait bound to generic.

```
struct MyStruct<T> {t:T}
unsafe impl<T> Send for MyStruct<T> {}
//fix: unsafe impl<T:Send> Send for MyStruct<T> {}
fn main() {
    let mut s = MyStruct { t:Rc::new(String::from("untouched_data")) };
    for _ in 0..99{
        let mut c = s.clone();
        std::thread::spawn(move || {
            if !Rc::get_mut(&mut c.t).is_none(){
                (*Rc::get_mut(&mut c.t).unwrap()).clear();
            }
            println!("c.t_=_{:?}", c.t);
        });
    }
}
```

Code 5 presents a PoC that implements the `Send` trait for `MyStruct` with a generic type `T`. However, it misses to bound `T` with the `Send` trait. As a result, users are able to construct a `MyStruct` object with `RC<T>`. Since `Rc<T>` is nonatomic, concurrently reading/writing its reference counter would suffer data race. Consequently, the counter value might be incorrect, leading to undefined behaviors when further mutating the wrapped value of `T`. In this PoC, the spawned thread might wrongly pass the uniqueness check and clear the wrapped string value. If the string buffer is already recycled due to data race, it would cause use-after-free or lead to a malformed stack. Most insufficient bound issues in our dataset are due to lack of `Send`/`Sync` bound, such as CVE-2020-35870, CVE-2020-35871, and CVE-2020-35886. The patches of such bugs simply bound `T` with `Sync` or `Send` traits.

Besides `Sync` or `Send` traits that are related to concurrent programs, there are also other bounds that may be missed. For example, CVE-2020-35901 and CVE-2020-35902 forget to restrict the generic `T` to `Pin<T>`, which means the memory location of `T` should be pinned; CVE-2020-35906 forgets to bound `T` with a static lifetime.

5.3.2 Generic Vulnerable to Specific Type. This type of bug also attacks the flexibility enabled by generic, but such bugs cannot be fixed simply by adding bound. Most bugs of this type in our dataset are std-lib bugs when handling special types, such as void (issue-48493) or zero-sized types (issue-42789 and 54857).

Code 6. PoC of unsound generic that does not respect the memory alignment.

```
#[repr(align(128))]
struct LargeAlign(u8);
struct MyStruct<T> { v:Vec<u8>,  _marker:PhantomData<*const T>, }
impl<T:Sized> MyStruct<T> {
    fn from(mut value:T) -> MyStruct<T> {
        let mut v = Vec::with_capacity(size_of::<T>());
        let src:*const T = &value;
        unsafe {
            ptr::copy(src, v.as_mut_ptr() as _, 1);
            v.set_len(size)
        }
        MyStruct { v, _marker:PhantomData }
    }
}
impl<T:Sized> ::std::ops::Deref for MyStruct<T> {
    type Target = T;
    fn deref(&self) -> &T{
        let p = self.v.as_ptr() as *const u8 as *const T;
        unsafe { &*p }
    }
}
fn main() {
    let s = MyStruct::from(LargeAlign(123));
    let v = &*s as *const _ as usize;
    assert!(v % std::mem::align_of::<LargeAlign>() == 0);
}
```

Code 6 presents a PoC based on CVE-2020-25796 and CVE-2020-35903. It defines a struct `MyStruct` with a generic `T`. `MyStruct` contains a constructor `from()` and an implemented dereference trait. To attack the generic, we define a new struct `LargeAlign(u8)` with a representation `#[repr(align(128))]` that requires 128-byte alignment. Ideally, when dereferencing `MyStruct` (line 24), the result is a value of type `LargeAlign` and should be 128-byte aligned. Unfortunately, the current implementation of `MyStruct` does not respect the alignment and fails the assertion.

5.3.3 Unsound Trait. Traits are abstract classes that can be derived or reimplemented by other types. The feature also leaves substantial room for users to explore attack code, i.e., deriving a safe trait (e.g., ,CVE-2020-25016 and CVE-2020-35866) or reimplementing the trait and causing undefined behaviors (e.g., CVE-2019-12083 and CVE-2020-25575) without using unsafe code.

Code 7 demonstrates a PoC based on a std-lib bug CVE-2019-12083. The PoC defines a trait `MyTrait` with a function `downcast()` that downcasts the trait to a particular type if `self.is::<T>()` evaluates to true. `self.is::<T>()` is based on another function `type_id()`, which is also implemented by the original trait and always returns a correct type. But `type_id()` could be reimplemented by the derived struct and may return a wrong type. As shown in line 21 for example, the reimplemented `type_id()` falsely returns a `u128` type for `u8`. As a result, it may falsely convert a boxed `u8` to `u128` and cause buffer over-read or overflow. Other examples of such type include CVE-2020-25575, CVE-2020-35889, and so on.

Code 7. PoC of unsound Trait.

```
trait MyTrait {
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<Self>()
    }
}
impl dyn MyTrait {
    pub fn is<T:MyTrait + 'static>(&self) -> bool {
        TypeId::of::<T>() == self.type_id()
    }
    pub fn downcast<T:MyTrait + 'static>(self: Box<Self>) -> Result<Box<T>, Box<dyn MyTrait>> {
        if self.is::<T>(){ unsafe {
           let raw:*mut dyn MyTrait = Box::into_raw(self);
           Ok(Box::from_raw(raw as *mut T))
        }} else { Err(self) }
    }
}
impl<T> MyTrait for Box<T> {}
impl MyTrait for u128 {}
impl MyTrait for u8 {
    fn type_id(&self) -> TypeId where Self: 'static {
        TypeId::of::<u128>()
    }
}
fn main(){
    let s = Box::new(10u8);
    let r = MyTrait::downcast::<u128>(s);
}
```

6 LESSONS LEARNED

Our analysis reveals that a major difference between Rust and other programming languages lies in the soundness promise of APIs. To this end, Rust compiler has already achieved soundness analysis for safe Rust, yet the main challenge lies in unsafe code. This section answers RQ-3 by discussing several suggestions or directions toward mitigating the problem raised by unsafe code.

Result Highlight for RQ-3: Our lessons learned are two-fold. Firstly, since some bugs demonstrate common patterns, we can summarize several best practices for assisting developers in avoiding unsoundness, such as those for generic bound declaration, avoiding bad drop at cleanup blocks, and function safety declaration. Secondly, since detecting some particular bugs related to unsafe code is not very difficult, we think extending the current static analysis scope of the compiler to support unsafe code should be a promising direction.

6.1 Best Practice for Code Suggestion

As some bugs in our dataset share common patterns, we can extract these patterns and make useful suggestions to developers for preventing them.

6.1.1 Generic Bound Declaration. The most common bugs in our dataset are those lacking Send or Sync bound to generics when implementing the Send or Sync trait for self-defined types or structs. 30 bugs in our dataset are of this pattern, such as CVE-2020-35886 and CVE-2020-35897.

Fixes of such bugs simply add Send or Sync bound. Therefore, we can recommend developers considering enforcing such bounds when implementing Send or Sync trait. We may develop a rule for detecting such issues automatically, i.e., if a Send or Sync trait is implemented for a type that has generic parameters, all the generic parameters should be bound to Send or Sync correspondingly. Note that Send or Sync trait are unsafe traits and implementing them are unsafe. If there are other unsafe traits in the future, implementing them may also suffer similar issues.

6.1.2 Avoiding Bad Drop at Cleanup Block. As discussed in Section 5.1.2, there are several similar patterns for the bugs related to the bad drop at cleanup blocks. A characteristic of these bugs is that there are vulnerable program points with shared mutable aliases or uninitialized memory, and the program control flow forks a cleanup branch in one of these points.

To deal with these bugs, our first suggestion is to avoid using some vulnerable APIs (e.g., `forget()` and `uninitialized()`) that are prone to memory-safety problems. For those APIs, Rust has already provided equivalent or replacing APIs which are safer to use, such as `ManuallyDrop<T>` for `forget()` and `MaybeUninit<T>` for `uninitialized()`. Both `ManuallyDrop<T>` and `forget()` can disable the automatic drop of some variables. However, `ManuallyDrop<T>` immediately takes effect when a variable is defined, while `forget()` can only be used afterwards. Note that if disabling automatic drop of some variables is a must, developers should do it as earlier as possible. Otherwise, panicking the program before reaching the disabling statement would be prone to dangling or invalid pointer issues during the stack unwinding process. Examples of such bugs are CVE-2019-15552, CVE-2019-15553, CVE-2019-16880, and CVE-2019-16881. Replacing `forget()` with `ManuallyDrop<T>` is a simple but effective way to fix such bugs. Since version 1.41.0 released in 2020, Rust has officially recommended using `ManuallyDrop<T>` instead of `forget()` in its manual (rustdoc). Developers should pay more attention if they insist on using `forget()`. Because the function borrows the ownership of its parameter value, which can prevent further mutating it, it is often impossible to call the function immediately after creating a new alias. This leaves chances to program panic before reaching `forget()`. Similarly, both `uninitialized()` and `MaybeUninit<T>` are APIs for creating variables with uninitialized memory. However, variables created via `MaybeUninit<T>` are treated as uninitialized until users have done the initialization by calling `assume_init()`, and Rust cleanup process does not drop such variables automatically before that. On the other hand, variables initialized with `uninitialized()` do not have such privilege. Due to this vulnerability, `uninitialized()` is deprecated by Rust since version 1.39.0 in 2019.

Our second suggestion is that when shrinking or expanding a buffer with unsafe code, the length of the resulting buffer should be set at an appropriate program point. In particular, if shrinking a buffer with `set_len()`, the `set_len()` function should be called as earlier as possible. Calling the function too late (e.g., CVE-2018-20991 and CVE-2020-25574) would lead to some program points with shared mutable aliases. If expending a buffer with `set_len()`, the function should be called after the new buffer has been fully initialized, or as late as possible. Otherwise, the new buffer may contain pointers that have not been fully initialized (e.g., CVE-2019-16138), leading to dropping invalid pointers. Note that Rust has another similar API (`resize()`) which is safer, but `set_len()` is preferred over `resize()` in some circumstances as it is low-level and has little performance overhead. Moreover, designing an algorithm to detect such bugs related to `set_len()` is also difficult in general because it requires understanding the program semantics or operations on the heap.

6.1.3 Function Safety Declaration. Several unsoundness issues in our dataset falsely declare unsafe functions as safe. As we know, interior unsafe plays an important role in determining the risk of unsoundness [15]. Can we make recommendations to developers based on whether their functions use unsafe code? In general, it is difficult, and our main recommendation is that if a

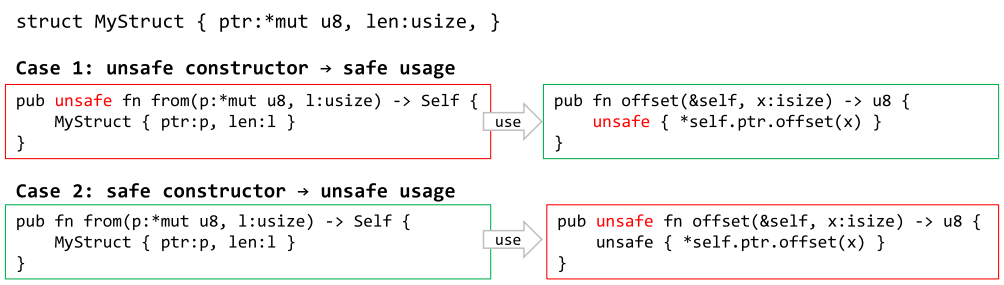


Fig. 6. Options of function safety declaration for structs with raw pointers.

function employs unsafe code unconditionally and it is not a member function, the function is likely to be declared as unsafe. Examples of such bugs that violate the rule include CVE-2019-15547 and CVE-2019-15548. Below we justify the point by discussing the four possible patterns between interior safety and function safety.

- *unsafe interior → unsafe function* which means a function that employs unsafe code is declared as unsafe. If the function contains unsafe, the function safety would depend on whether the dangerous inputs have been naturalized before reaching the unsafe code. Without such guarantee, the function should be declared as unsafe by default, such as `offset()` in Case 2 of Figure 6.
- *unsafe interior → safe function* which means a function that employs unsafe code is declared as safe. This is the most popular pattern of our function safety declaration bugs. Can we make recommendations based on whether the function employs some restrictions before reaching its unsafe code? e.g., if the first statement is unsafe, does it imply the function should be declared as unsafe? Unfortunately, this is not true, especially for structures. Taking `offset()` in Case 1 of Figure 6 as an example, it directly dereferences the raw pointer of `self.ptr`, but it is not a bug because its constructor is declared as unsafe. If the constructor can guarantee that these member values meet safety requirements, then its member functions employ unsafe code directly can also be declared as safe.
- *safe interior → safe function* which means a function with pure safe code is defined as safe. If a function does not involve unsafe code internally, it is true that using the function in any way would not cause memory-safety bugs, such as `from()` in Case 2 of Figure 6.
- *safe interior → unsafe function* which means a function with pure safe code is defined as unsafe. Although it might be strange, this pattern also exists in practice, such as `from()` in Case 1 of Figure 6 or when implementing unsafe traits.

Note that Case 1 and Case 2 of Figure 6 are two typical patterns for designing structs with raw pointers. Case 1 declares the constructor as unsafe, and therefore all other member functions can be declared safe. On the other hand, Case 2 declares the constructor as safe and other member functions as unsafe. Both cases are employed in real-world Rust projects. Based on these cases, we can make another recommendation that if a struct contains members of raw pointers, these members should not be exposed as public to avoid unsound construction by directly assign the raw pointer (e.g., CVE-2020-36205) without using any unsafe marker. Otherwise, all other member functions would be risky to unsoundness.

6.2 Static Analysis with Unsafe Code

According to the memory-safety promise of Rust, developers should make sure their provided APIs do not incur undefined behaviors no matter how they are employed. However, preventing general

memory-safety bugs via static analysis is hard. According to Rice's theorem [32], it is impossible to have a general algorithm for this problem that is sound, complete, and can terminate. Therefore, Rust compiler makes a tradeoff by targeting the soundness analysis of safe code only, leaving the responsibility of employing unsafe code to developers. However, there are also some interesting culprits worth further investigation, especially those related to the side effect of automatic memory reclaim.

To mitigate the problem of automatic memory reclaim, we may perform alias analysis in the MIR level, and then perform use-after-drop, double drop, or dropping uninitialized memory detection via abstract interpretation [8]. Note that although general pointer analysis is hard, this problem can be simplified in two ways. Firstly, considering only the pointers related to unsafe constructors can simplify the analysis problem and should still be effective for most bugs in our dataset. Secondly, there are two approximation directions for solving such problems, the first direction is to perform may-alias-then-check, i.e., to maintain an alias pool recording the may-alias relationship between the source buffer and the newly constructed variable (e.g., via inclusion-based pointer analysis [2]) and then check whether a potential bug caused by aliases is spurious. The second direction is to perform precise alias analysis but in a conservative mode, e.g., terminate after some threshold. Taking Code 1 as an example, the returned value v contains an alias of s, both of which are mutable aliases. If the compiler knows the two pointers could be aligned, it may either report the bug or transfer the ownership from s to v and avoid dropping s. We have done an initial exploration of this problem with a path-sensitive pointer analysis approach. It employs a modified Tarjan algorithm to optimize the path-sensitive analysis problem by removing redundant paths. Meanwhile, it handles inter-procedural calls recursively with a cached-based approach for reusing the aliases generated by each callee. Our detailed approach is described in another work [9]. Based on the approach, we have developed a prototype and integrated it into Rust compiler. Experimental results show that the tool is effective in detecting related CVEs, and we have also detected several new bugs in this pattern. Considering the potential overhead and false-positive issues for practical usage, we think a promising direction in the future might be integrating the feature into a specific analysis tool of Rust IDE that is independent to Rust compiler, such as rust-analyzer[12] that interacts with the editor via the language server protocol [27].

Another sophisticated challenge for Rust bug detection is that some APIs are only unsound and they do not accomplish bad behaviors by themself. In order to detect these unsoundness issues, one possible direction is to employ automated use case generation based on API dependency graph traversal and then perform analysis based on each use case [20], or to leverage a contract-based approach [7] and then detect whether there are violations of specified contracts. We leave such investigations as our future work.

7 THREATS TO VALIDITY

Our work performs an in-depth study of memory-safety bugs found in real-world Rust programs. It is mainly based on a dataset of 186 memory-safety bugs. Although the dataset may not be very large, it contains all Rust CVEs with memory-safety issues by 2020-12-31, which are the most severe bugs. Therefore, our analysis result should be representative. Moreover, because our study approach is systematic and in-depth, the dataset is enough for justifying the derived findings and suggestions. As the dataset of bugs may get further enriched in the future, there could be new patterns of culprits found, and the categories employed in Section 5 could be further extended.

[12] https://rust-analyzer.github.io/.

8 RELATED WORK

8.1 Comparison with Qin's Work

There are four other independent articles that have similar purposes as we do, including [4, 15, 30, 39]. Both Evans et al. [15] and Astrauskas et al. [4] have performed a large-scale study regarding the usage of unsafe code, but without bug analysis. The articles [30, 39] are produced by the same team, and [39] is a preprint version. Therefore, we mainly compare the difference of our article with [30].

The preprint version [39] by Yu et al. studies 18 concurrency bugs from three Rust projects, and [30] by Qin et al. extends the work with more bugs and expands the analysis scope from concurrency issues to both memory-safety bugs and concurrency bugs. They categorize 70 memory-safety bugs into four patterns: (1) safe, (2) unsafe, (3) safe->unsafe, and (4) unsafe->safe, and discuss several typical bugs associated with each category. In comparison, our work focuses on memory-safety bugs, and our dataset contains 186 memory-safety bugs, which is much larger than [30]. Note that [30] also studies many concurrency bugs, and most of these bugs do not incur memory-safety issues but only deadlock or logical errors that crash the program. Since our work considers more memory-safety bugs, it extracts more representative bug patterns than [30], such as the sub-categories of unsound generic or trait. Furthermore, we focus on memory-safety bugs only and perform an in-depth study regarding these bugs, and we finally obtain several novel findings beyond [30], such as the challenges for enforcing soundness when employing generic and trait. In short, while [30] provides more understandings of concurrency issues than us, our work provides substantial extensions to the result of [30] in memory-safety bugs.

8.2 Other Work

While some work shares experience in using Rust, such as [3, 23, 24, 28], there are also several articles that focus on the reliability of Rust programs [25, 36] and the Rust language system [5, 6, 21].

A majority of existing articles on Rust reliability focuses on formal verification [5, 11, 21] and bug detection [6, 13, 25, 36]. Formal verification aims at proving the correctness of Rust programs mathematically. RustBelt [11, 21] is a representative work in this direction. It defines a set of rules to model Rust programs and employs these rules to prove the security of Rust APIs. It has verified that the basic typing rules of Rust are safe, and any closed programs based on well-typed Rust should not exhibit undefined behaviors. Astrauskas et al. [5] proposed a technique that can assist developers to verify their programs with formal methods. Dewey et al. [13] proposed a fuzzing testing approach to detect the bugs of Rust programs. Lindner et al. [25] proposed to detect panic issues of Rust programs via symbolic execution. Besides, some work focuses on the unsafe part of Rust and studies how to isolate unsafe code blocks (e.g., [1, 22, 26]). We will not go into further details because the purposes of these articles are quite different from our work.

9 CONCLUSION

This work studies the effectiveness of Rust in fighting against memory-safety bugs. We have manually analyzed the culprits and consequences of 186 memory-safety bugs, covering all Rust CVEs with memory-safety issues. Our study results reveal that Rust successfully limits the risks of memory-safety issues in the realm of unsafe code, and the magic lies in the soundness requirement of safe APIs. As a result, many memory-safety bugs in our dataset are mild soundness issues. Furthermore, we derive three typical categories of memory-safety bugs in Rust, including automatic memory reclaim, unsound function, and unsound generic or trait. In particular, auto memory reclaim uncovers the side effect of Rust OBRM, unsound function unveils the essential

challenge of Rust development for avoiding unsoundness, and unsound generic or trait reveals the advanced challenge for developers to ensure soundness when using polymorphism and inheritance. Furthermore, our analysis leads to some suggestions for improving the security of Rust development. For program developers, we provide several recommendations concerning the best practices in using some APIs. For compiler/tool developers or researchers, our suggestion is to consider extending the current program analysis mechanism from safe code to supporting some typical bugs caused by unsafe code, especially those related to the automatic memory reclaim scheme. We hope our work can raise more discussions regarding the memory-safety effectiveness of Rust and inspire more ideas toward improving the resilience of Rust.

ACKNOWLEDGMENTS

We thank all the hunters of bugs employed in our dataset for their contributions in reporting these bugs and anonymous reviewers of the paper for their constructive comments.

REFERENCES

[1] Hussain M. J. Almohri and David Evans. 2018. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*. 248–255.

[2] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language*. Ph.D. Dissertation. University of Cophenhagen.

[3] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the servo web browser engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*. 81–89.

[4] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe rust? In *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[5] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging rust types for modular specification and verification. In *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[6] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying rust programs with SMACK. In *Proceedings of the International Symposium on Automated Technology for Verification and Analysis*. Springer, 528–535.

[7] Albert Benveniste, Benoît Caillaud, Alberto Ferrari, Leonardo Mangeruca, Roberto Passerone, and Christos Sofronis. 2007. Multiple viewpoint contract-based specification and design. In *Proceedings of the International Symposium on Formal Methods for Components and Objects*. Springer, 200–225.

[8] Patrick Cousot. 1996. Abstract interpretation. *ACM Computing Surveys* 28, 2 (1996), 324–328.

[9] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2021. SafeDrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. arXiv:2103.15420 Retrieved from https://arxiv.org/abs/2103.15420.

[10] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 2018. The benefits and costs of writing a POSIX kernel in a high-level language. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 89–105.

[11] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt meets relaxed memory. In *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[12] Thurston H. Y. Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 555–566.

[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2015. Fuzzing the rust typechecker using CLP (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 482–493.

[14] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. Heaphopper: Bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX Security Symposium*. 99–116.

[15] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In *Proceedings of the 42nd International Conference on Software Engineering*. IEEE.

[16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 1–13.

[17] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[18] Tais B. Ferreira, Rivalino Matias, Autran Macedo, and Lucio B. Araujo. 2011. An experimental study on memory allocators in multicore and multithreaded applications. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 92–98.

[19] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical cache attacks on the MMU. In *Proceedings of the Network and Distributed System Security Symposium*, Vol. 17. 26.
[20] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In *Prof. of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE'21)*.
[21] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the rust programming language. In *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.
[22] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic sandboxing of unsafe components in rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*. 51–57.
[23] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: Experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. 21–26.
[24] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Shane Leonard, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The tock embedded operating system. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–2.
[25] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! verification of rust programs by symbolic execution. In *Proceedings of the IEEE 16th International Conference on Industrial Informatics*. IEEE, 108–114.
[26] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with xrust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 234–245.
[27] Mónika Mészáros, Máté Cserép, and Anett Fekete. 2019. Delivering comprehension features into source code editors through LSP. In *Proceedings of the 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, 1581–1586.
[28] Kai Mindermann, Philipp Keck, and Stefan Wagner. 2018. How usable are rust cryptography APIs?. In *Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 143–154.
[29] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type inference with constrained types. *Theory and Practice of Object Systems* 5, 1 (1999), 35–55.
[30] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practicesand issues in real-world rust programs. In *Proceedings of the of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.
[31] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. 2012. A mechanized semantics for C++ object construction and destruction, with applications to resource management. *ACM SIGPLAN Notices* 47, 1 (2012), 521–532.
[32] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society* 74, 2 (1953), 358–366.
[33] Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. 2005. The impact of software engineering research on modern progamming languages. *ACM Transactions on Software Engineering and Methodology* 14, 4 (2005), 431–477.
[34] Bjarne Stroustrup. 2001. Exception safety: Concepts and techniques. In *Proceedings of the Advances in Exception Handling Techniques*. Springer, 60–76.
[35] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*. IEEE, 48–62.
[36] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: A bounded verifier for rust (n). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 75–80.
[37] Xiangrong Wang, Jun Xu, and Christopher H. Pham. 2004. An effective method to detect software memory leakage leveraged from neuroscience principles governing human memory behavior. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*. IEEE, 329–339.
[38] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium*. 781–797.
[39] Zeming Yu, Linhai Song, and Yiying Zhang. 2019. Fearless concurrency? understanding concurrent programming safety in real-world rust software. arXiv:1902.01906 Retrieved from https://arxiv.org/abs/1902.01906.

Received September 2020; revised May 2021; accepted May 2021