12-2023

A closer look at the security risks in the Rust ecosystem

Xiaoye ZHENG

Zhiyuan WAN

Yun ZHANG

Rui CHANG

David LO
Singapore Management University, davidlo@smu.edu.sg

Citation

ZHENG, Xiaoye; WAN, Zhiyuan; ZHANG, Yun; CHANG, Rui; and LO, David. A closer look at the security risks in the Rust ecosystem. (2023). *ACM Transactions on Software Engineering and Methodology*. 33, (2), 1-34.

Available at: https://ink.library.smu.edu.sg/sis_research/8644

A Closer Look at the Security Risks in the Rust Ecosystem

XIAOYE ZHENG, Zhejiang University, China
ZHIYUAN WAN*, Zhejiang University, China
YUN ZHANG, Hangzhou City University, China
RUI CHANG, Zhejiang University, China
DAVID LO, Singapore Management University, Singapore

Rust is an emerging programming language designed for the development of systems software. To facilitate the reuse of Rust code, `crates.io`, as a central package registry of the Rust ecosystem, hosts thousands of third-party Rust packages. The openness of `crates.io` enables the growth of the Rust ecosystem but comes with security risks by severe security advisories. Although Rust guarantees a software program to be safe via programming language features and strict compile-time checking, the `unsafe` keyword in Rust allows developers to bypass compiler safety checks for certain regions of code. Prior studies empirically investigate the memory safety and concurrency bugs in the Rust ecosystem, as well as the usage of `unsafe` keywords in practice. Nonetheless, the literature lacks a systematic investigation of the security risks in the Rust ecosystem.

In this paper, we perform a comprehensive investigation into the security risks present in the Rust ecosystem, asking "what are the characteristics of the vulnerabilities, what are the characteristics of the vulnerable packages, and how are the vulnerabilities fixed in practice?". To facilitate the study, we first compile a dataset of 433 vulnerabilities, 300 vulnerable code repositories, and 218 vulnerability fix commits in the Rust ecosystem, spanning over 7 years. With the dataset, we characterize the types, life spans, and evolution of the disclosed vulnerabilities. We then characterize the popularity, categorization, and vulnerability density of the vulnerable Rust packages, as well as their versions and code regions affected by the disclosed vulnerabilities. Finally, we characterize the complexity of vulnerability fixes and localities of corresponding code changes, and inspect how practitioners fix vulnerabilities in Rust packages with various localities.

We find that memory safety and concurrency issues account for nearly two thirds of the vulnerabilities in the Rust ecosystem. It takes over 2 years for the vulnerabilities to become publicly disclosed, and one third of the vulnerabilities have no fixes committed before their disclosure. In terms of vulnerability density, we observe a continuous upward trend at the package level over time, but a decreasing trend at the code level since August 2020. In the vulnerable Rust packages, the vulnerable code tends to be localized at the file level, and contains statistically significantly more unsafe functions and blocks than the rest of the code. More popular packages tend to have more vulnerabilities, while the less popular packages suffer from vulnerabilities for more versions. The vulnerability fix commits tend to be localized to a limited number of lines of code. Developers tend to address vulnerable safe functions by adding safe functions or lines to them, vulnerable unsafe blocks by removing them, and vulnerable unsafe functions by modifying unsafe trait implementations. Based on our findings, we discuss implications, provide recommendations for software practitioners, and outline directions for future research.

CCS Concepts: • **Software and its engineering → Language types**; • **Human-centered computing → Open source software**; • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Rust, ecosystem, security risks, vulnerability, empirical study

1 INTRODUCTION

Modern software systems benefit from reusing code from open source projects, leading to the formation of complex interdependency networks, i.e., software ecosystems [60]. The reusable code

*Corresponding author.

Authors' addresses: Xiaoye Zheng, Zhejiang University, Hangzhou, China, xiaoyez@zju.edu.cn; Zhiyuan Wan, Zhejiang University, Hangzhou, China, wanzhiyuan@zju.edu.cn; Yun Zhang, Hangzhou City University, Hangzhou, China, yunzhang@zucc.edu.cn; Rui Chang, Zhejiang University, Hangzhou, China, crix1021@zju.edu.cn; David Lo, Singapore Management University, Singapore, davidlo@smu.edu.sg.

usually takes the form of packages delivered by package management systems, such as npm for JavaScript packages, PyPI for Python packages, and Maven for Java packages. In recent years, researchers conduct substantial studies to investigate a variety of aspects of software ecosystems, including their evolution [17, 22], dependencies of packages [14–16] and security risks [1, 27, 71]. A few studies make comparisons across software ecosystems, such as the structure [26] and evolution [17] of dependencies across software ecosystems.

Rust is an emerging programming language designed for the development of systems software [10, 49, 56]. Over the past few years, Rust has experienced explosive growth and gained popularity [33–35], especially in developing systems software like operating systems and browsers [40, 43, 48, 54, 57]. According to the annual developer survey of Stack Overflow[1], Rust has been named the "most loved programming language" for six years in a row, from 2016 to 2021. To support Rust practitioners with third-party code, `crates.io`, as a central package registry of the Rust ecosystem, provides thousands of reusable packages (crates). The openness of `crates.io` enables the growth of the Rust ecosystem, ranging from small utility packages to complex Web programming frameworks and cryptography libraries. Rust guarantees a software program to be safe via programming language features, and with strict compile-time checking [21, 51, 55]. Nonetheless, the openness of the Rust ecosystem comes with security risks as evidenced by severe security advisories. For instance, in January 2022, Rust maintainers released a security update for a high-severity vulnerability (CVE-2022-21658). Attackers could abuse the vulnerability to purge files and directories from a vulnerable system in an unauthorized manner. In addition, Rust introduces an `unsafe` keyword that allows developers to bypass compiler safety checks for certain regions of code. It is unclear if the code regions with `unsafe` keywords tend to suffer from more vulnerabilities.

Several recent works perform empirical studies to characterize memory safety and concurrency bugs in Rust systems [39, 65] and understand the usage of `unsafe` keyword in the Rust ecosystem [2, 19]. Nevertheless, the literature lacks a systematic investigation of the security risks of the Rust ecosystem. Given the popularity of Rust, a better understanding of its security risks is an important step toward sustaining and securing this software ecosystem. To address this gap, we followed a mixed-methods approach to perform a large-scale empirical study on the vulnerabilities of the Rust ecosystem.

We compiled a dataset of 433 vulnerabilities, 300 vulnerable code repositories, and 218 vulnerability fix commits in the Rust ecosystem, spanning over 7 years in history. With our dataset, we investigated the following research questions:

RQ1: What are the characteristics of the vulnerabilities in the Rust ecosystem?

Previous studies investigated the characteristics of specific types of vulnerabilities in the Rust ecosystem, e.g., memory safety [65] and concurrency issues [39]. The answer to this question aims to build a systematic view of a wide range of vulnerabilities in the Rust ecosystem, rather than a focused view of specific vulnerability types.

In RQ1, we characterized the types, life spans, and evolution of numbers of vulnerabilities in the Rust ecosystem. Our study identified 17 types of vulnerabilities disclosed in the Rust ecosystem, among which memory safety and concurrency issues account for two-thirds of the categorized vulnerabilities and demonstrate the fastest growth rates over time. It takes an average of 770 days (2.1 years) for a vulnerability to be disclosed after its introduction in a code repository. One-third of the vulnerabilities have no fixes released by their public disclosure, leaving a window of opportunity for potential attacker exploitation. The number of vulnerabilities disclosed grows slowly from November 2014 to November 2020, and has experienced two rapid growth phases, starting from November 2020 and July 2021, respectively. Meanwhile, the number of vulnerabilities

[1] https://insights.stackoverflow.com/survey/2021#technology-most-loved-dreaded-and-wanted

introduced into code repositories demonstrates a linear growth from July 2015 to January 2020, and has stabilized since March 2020. In addition, the normalized numbers of vulnerabilities per one thousand packages and lines of code indicate an increasing trend in package-level security risks but a decreasing trend in code-level security risks, respectively.

RQ2: What are the characteristics of the vulnerable packages in the Rust ecosystem?

Package reuse in software ecosystems introduces potential security risks that arise from vulnerable packages and propagate through multiple levels of dependencies among packages [26, 71]. A prior study [19] investigated the usage of `unsafe` keyword in Rust packages and found that the security risks of limited usage of `unsafe` keyword could be amplified by their propagation through package dependencies. Nonetheless, it remains unclear whether unsafe code introduces more vulnerabilities compared to safe code in Rust packages. The answer to this question aims to guide practitioners in securing the Rust ecosystem.

In RQ2, we investigated the affected versions, popularity, categorization, and affected code regions of vulnerable Rust packages. Our study found that the vulnerable packages in the Rust ecosystem have an average of 1.3 disclosed vulnerabilities and 28.6 versions affected by the vulnerabilities. Popular packages tend to have more vulnerabilities, while unpopular packages tend to have more versions affected. The *memory management* package category has the greatest number of vulnerabilities per package among different Rust package categories, and tends to have more *memory access*, *memory management*, and *synchronization* vulnerabilities as compared to other package categories. In terms of vulnerability locality, a disclosed vulnerability affects 1.85 files, 3.35 safe functions, 0.15 unsafe functions, and 1.39 unsafe blocks on average in the vulnerable packages. 95% of the affected functions are safe functions. Among the affected safe functions, 41.5% contain unsafe blocks in their body. In the vulnerable packages, vulnerable code has statistically significantly higher ratios of unsafe functions and unsafe blocks compared to complete code, implying the potential higher security risks in unsafe functions and unsafe blocks.

RQ3: How are the vulnerabilities in the Rust ecosystem fixed in practice?

While numerous works have investigated general vulnerability fixes [37, 50, 52, 69], few have considered the vulnerability fixes in the Rust ecosystem. The characteristics of vulnerability fixes are important to understand as they may reflect the ability to expeditiously generate fixes, verify their safety, and assess their impact on applications [27].

In RQ3, we considered the facets of vulnerability fixes such as the complexity of fixes and locality of code changes, and investigated how practitioners fix vulnerabilities in Rust packages with different localities. The study revealed that the commits of vulnerability fixes involve an average of 41 and 18 LOC added and deleted, touching 3.85 safe functions, 0.16 unsafe functions, and 1.49 unsafe blocks on average. 96% of the touched functions are safe functions, among which, 38.8% contain unsafe blocks in their body. The vulnerabilities of different types differ widely in localities of fix commits, among which, the *exception management* vulnerabilities have the greatest number of safe functions touched by their fix commits. This indicates that their fixes are the least localized at the function level, indicating potential challenges of fixing in practice.

In addition, our study uncovered three patterns in the vulnerability fixes – developers tend to (1) add safe functions or add lines in safe functions to fix vulnerable safe functions, (2) remove unsafe blocks to fix vulnerable unsafe blocks, and (3) modify unsafe trait implementations[2] to fix vulnerable unsafe functions.

Based on our findings, we discuss implications and provide practical lessons for securing the Rust ecosystem, such as undertaking comparable efforts into safe and unsafe code when securing Rust packages. We also highlight several research avenues, such as continuous collection and analyses

[2]The trait syntax in Rust is similar to the Java interface syntax.

of vulnerabilities to increase the awareness of security risks in the Rust ecosystem. This paper makes the following contributions:

- We performed a large-scale empirical study to investigate the security risks in the Rust ecosystem.
- We provided a dataset that include 433 vulnerabilities, 300 vulnerable code repositories, and 218 vulnerability fix commits for future investigations by others[3].
- We summarized the vulnerability fix patterns of different localities in Rust code, which can be used as guidelines to resolve vulnerabilities in practice.
- We provided a discussion of practical implications and outlined future avenues of research.

The replication package is online at https://github.com/ZXXYy/rust_ecosystem.

2 BACKGROUND AND PRELIMINARY EXPERIMENTS

This section gives some background on Rust, including its safety mechanisms and unsafe Rust code that are relevant to our study, and conducts preliminary experiments on Rust ecosystem.

2.1 Rust Safety Mechanisms

Rust is a type-safe language designed for systems software development, which gives developers low-level control over resources but ensures memory and thread safety via a set of strict rules. The Rust compiler checks these rules to statically rule out potential safety issues. Rust programs behave like C programs, and could achieve comparable runtime performance as C programs. The Rust's safety mechanisms aim to prevent memory and thread safety issues that have plagued C programs. The safety mechanisms center around several basic concepts:

- **Ownership.** The ownership mechanism governs how a Rust program manages its memory, and prevents a Rust program from reading uninitialized memory and dangling pointers. Under Rust's basic ownership rule, a value (memory location) has one exclusive owner (variable). When the owner of a value goes out of a specific scope, the value would be dropped or freed. The variable assignment leads to the transfer of ownership. Once a variable loses the ownership of a value, the variable would become unusable.
- **Borrowing.** To enable sharing a value without moving its ownership, the borrowing mechanism allows the creation of a reference and passes the reference to another variable. In addition, Rust supports multiple shared immutable references, i.e., references that allow read-only aliasing. Rust enforces the memory locations reachable by a shared reference to be immutable to prevent data races and inadvertent side effects.
- **Lifetime.** Lifetime explains the scopes for which references in a Rust program are valid. The lifetime feature in Rust includes a variety of generics that indicate how references relate to each other. Specifically, to determine when references go out of their scopes, the compiler associates each borrowed reference with a lifetime and tracks constraints between references. The lifetime inference assures that the lifetime of a borrowed ownership would last long enough for use.

2.2 Unsafe Rust Code

Rust developers usually need flexibility in writing their code, including accessing arbitrary memory with C-style pointers, invoking system calls, and accessing global static memory. Rust allows programs to bypass its security mechanisms with the `unsafe` keyword. Code regions marked with the `unsafe` keyword could bypass Rust's compiler checks, and be able to perform five types of operations: dereferencing and manipulating raw pointers, calling unsafe functions, accessing or

[3] https://zenodo.org/record/7828059#.ZDo1v-xBy3Y

modifying mutable static variables (i.e., global variables), implementing unsafe traits, and accessing fields of `unions`. For simplicity, we use the phrase "unsafe code" throughout the paper to refer to the code regions that are marked with the `unsafe` keyword. The code regions that can be marked as unsafe include:

- **Unsafe Blocks.** An unsafe block defines a block of Rust code in which some compiler safety checks would be disabled. For instance, as shown in Listing 1, an unsafe block dereferences a raw pointer `r`. The dereferencing operation can bypass compiler checks due to the `unsafe` keyword. Note that if a block of Rust code is marked with the `unsafe` keyword but does not contain any of the aforementioned five types of operations, the compiler would emit a warning message.

Listing 1. Unsafe block example.

```
let mut address = 5;
// create a raw pointer
let r = &address as *const i32;
unsafe {
    print!("r␣is:␣{}", *r)
}
```

- **Unsafe Functions.** A Rust function can be declared as an unsafe function with the `unsafe` keyword. The `unsafe` keyword requires the callers of unsafe functions to satisfy some preconditions or bypass compiler checks via unsafe blocks. If an unsafe function only includes safe operations, the compiler would not emit a warning because it cannot tell whether programmers do it intentionally or by mistake. Listing 2 shows a typical usage of an unsafe function: the unsafe function `bar()` is called within an unsafe block of the safe function `foo()`, indicating that unsafe functions are encapsulated by their callers.

Listing 2. Unsafe function example.

```
unsafe fn bar() {...}

fn foo() { // a safe function
    unsafe {
        bar(); // call an unsafe function in an unsafe block
    }
}
```

- **Unsafe Traits**. The trait is an advanced feature in the Rust type system to enable inheritance. In general, traits of Rust are similar to interfaces to Java or abstract classes to C++. A trait can be declared as unsafe with the `unsafe` keyword if it contains unsafe functions or its implementations is required to satisfy any invariant.

2.3 Preliminary Investigation of Rust Ecosystem

Rust is a striving ecosystem with ongoing and even accelerating growth in the number of packages and downloads. An increasing number of areas start to choose Rust as the programming language for software development. Figure 1 shows the evolution of the number of packages in the Rust ecosystem since its inception. The first package in the Rust ecosystem hosted on `crates.io` was published on November 11, 2014. The number of packages grows 1.6x per year on average from 2015 to 2020. From 2020 to 2021, the growth in the number of packages slightly slows down, exhibiting a 1.4x growth rate. Figure 2 shows the number of packages that are being created on `crates.io` every month since November 2014. From late 2014 to early 2018, the increasing number

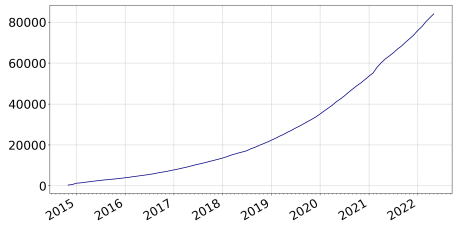


Fig. 1. Evolution of number of packages per month.

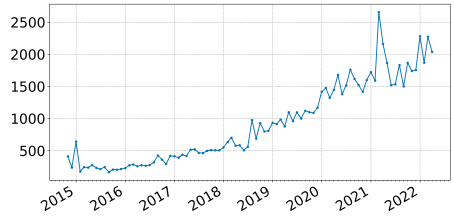


Fig. 2. Growth rate of packages created per month.

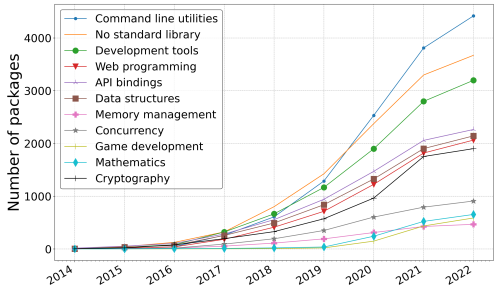


Fig. 3. Evolution of numbers of packages per year across package categories.

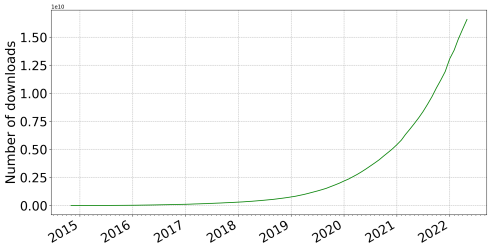


Fig. 4. Evolution of package downloads.

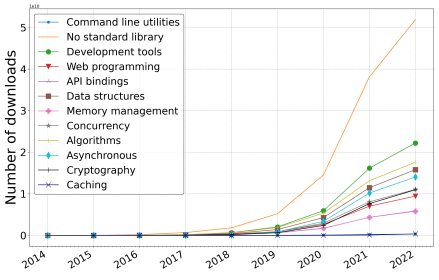


Fig. 5. Evolution of package downloads across package categories.

of packages per month is less than 500. Since early 2018, the growth in the increasing number of packages per month accelerates, experiencing a peak in March 2021, which may be due to the official announcement of the Rust Foundation[4] on February 8, 2021. Following the peak, the increasing number of packages per month shows steady growth and resembles the trend prior to 2021, indicating that the announcement acted as a boost for the Rust ecosystem. As shown in Figure 3, the top 5 categories with the most packages, *command line utilities*, *no standard library*, *development tools*, *api bindings* and *data structures*, undergo continuous near-exponential growth in the number of packages over time, which resembles the growth of the Rust ecosystem.

We also investigate the downloads of all Rust packages on a monthly basis from November 2014 to May 2022. As Figure 4 shows, the downloads of Rust packages grow exponentially from November 2014 to May 2022. The growth rate of downloads has experienced a dramatic increase since April

[4]https://foundation.rust-lang.org/news/2021-02-08-hello-world/

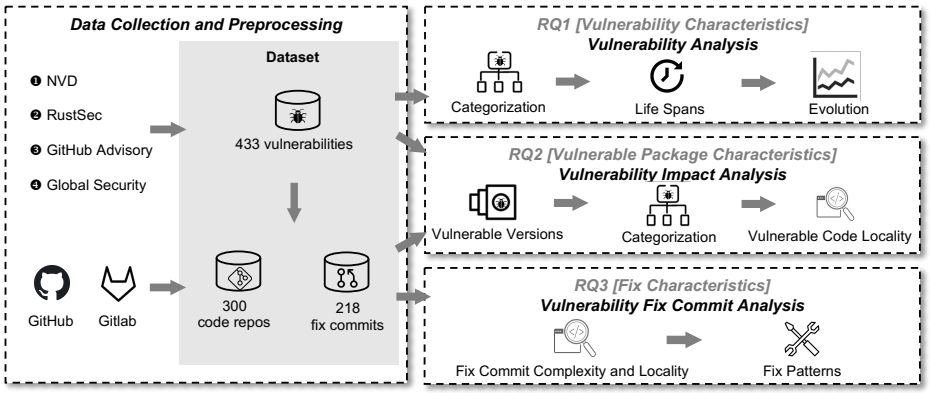


Fig. 6. Overview of research methodology.

2019, which is far greater than the growth rate of package numbers. Figure 5 presents the top 5 categories of Rust packages that have the greatest downloads of packages, i.e., *no standard library*, *development tools*, *algorithms*, *data structures* and *asynchronous*. The 5 categories of Rust packages experience a continuous near-exponential growth in downloads over time, resembling the growth of package downloads in the Rust ecosystem. The number of packages in the *memory management* and *concurrency* categories grow linearly as indicated by regression analysis (R^2 = 0.9163 and 0.9460).

3 RESEARCH METHODOLOGY

We designed and conducted a mixed-methods empirical study, analyzing a dataset of vulnerabilities, vulnerable packages, and vulnerability fixes in the Rust ecosystem, as depicted in Figure 6. Our research methodology is detailed in the following subsections.

3.1 Data Collection and Preprocessing

Step 1: Collecting vulnerabilities in the Rust ecosystem. We collected an initial set of 776 vulnerabilities disclosed on OSV[5] from November 11, 2014, to May 24, 2022. OSV is a distributed vulnerability database for open source ecosystems, which serves as an aggregator of vulnerability databases including GitHub Security Advisories, RustSec, the National Vulnerability Database (NVD), and the Global Security Database. We identified 343 duplicated vulnerabilities from the initial set and merged the information of vulnerabilities with the same references to NVD or RustSec, resulting in a final set of 433 unique vulnerabilities. For each vulnerability, we obtained the summary, detail, published date, modified date, vulnerability introduced and fixed versions, references (e.g., code repository, fix commit, issue, and pull request), and type(s) of the vulnerability.
Step 2: Locating code repositories of vulnerable Rust packages. We further located the code repositories of vulnerable Rust packages by following the references provided by disclosed vulnerabilities. We found that 17 disclosed vulnerabilities from 13 Rust packages do not provide references to their code repositories, thus we did not consider the 17 vulnerabilities in this step. As a result, we obtained a total of 300 code repositories for vulnerable packages on GitHub and GitLab.

[5]https://osv.dev

```
@@ -8,7 +8,7 @@
-use alloc::vec::Vec;
+use alloc::boxed::Box;
 use core::cell::UnsafeCell;
 use core::fmt;
 use core::marker::PhantomData;

@@ -110,7 +110,7 @@ impl<T> ArrayQueue<T> {
         // Allocate a buffer of `cap` slots initialized
         // with stamps.
         let buffer = {
-            let mut v: Vec<Slot<T>> = (0..cap)
+            let mut boxed: Box<[Slot<T>]> = (0..cap)
                 .map(|i| {
                     // Set the stamp to `{ lap: 0, index: i }`.
                     Slot {

@@ -119,8 +119,8 @@ impl<T> ArrayQueue<T> {
                     }
                 })
                 .collect();
-            let ptr = v.as_mut_ptr();
-            mem::forget(v);
+            let ptr = boxed.as_mut_ptr();
+            mem::forget(boxed);
             ptr
         };

@@ -425,7 +425,11 @@ impl<T> Drop for ArrayQueue<T> {
         // Finally, deallocate the buffer, but don't run any destructors.
         unsafe {
-            Vec::from_raw_parts(self.buffer, 0, self.cap);
+            // Create a slice from the buffer to make
+            // a fat pointer. Then, use Box::from_raw
+            // to deallocate it.
+            let ptr = core::slice::from_raw_parts_mut(self.buffer, self.cap) as *mut [Slot<T>];
+            Box::from_raw(ptr);
         }
     }
 }
```

Fig. 7. An example fix commit applied to vulnerability *RUSTSEC-2020-0052* of `crossbeam-channel`.

Step 3: Identifying vulnerability-fix commits in vulnerable code repositories. We identified an initial set of 287 vulnerability-fix commits by analyzing three types of fix references that are provided by collected vulnerability reports:

- *Commit.* We considered the commit in the reference as the vulnerability-fix commit.
- *Pull Request.* Given a pull request could have multiple commits, we identified vulnerability-fix commits by searching vulnerability-fix related keywords in commit messages, including fix, repair, error, bug, issue, exception and cve.
- *Issue.* We located the pull requests or commits related to an issue to identify the vulnerability-fix commits. We only considered the closed issues because they indicate that the corresponding vulnerabilities are fixed.

With the initial set of vulnerability-fix commits, we further excluded commits that do not have code for vulnerability fixing, .e.g., for refactoring purpose. Specifically, we inspected the vulnerability-fix commits in the initial set and excluded 69 commits irrelevant to vulnerability fixes, including 34 commits that modified change logs, corrected spelling or styling, 28 that reported the packages as unmaintained and did not fix any vulnerabilities, and 7 that involved code refactoring in the fix. As a result, we collected 218 vulnerability-fix commits for 180 vulnerabilities in the Rust ecosystem.

3.2 Processing Vulnerability Fixes

For each fix commit we collected, we considered the removed lines in the fix commit as *vulnerable code*, and the added lines as *fixing code*, as with prior work [9, 42, 50, 59]. Similar to prior work [27], we excluded non-source code files, e.g., documentation, change logs, and test files, and further

removed non-functional source code, e.g., empty lines, comment lines, and lines that are not inside any functions. We used git diff wrapped in PyDriller [53] to obtain the textual diffs of the commit and located the removed and added lines. Figure 7 gives an example of a fix commit that represents code changes as textual diffs. The black, red and green colors represent unchanged code, deleted lines, and added lines, respectively. In the example fix commit, lines 12, 22, 23, and 32 are vulnerable code; lines 13, 24, 25, and 36 are fixing code; and lines 2, 3, 33, 34, and 35 are non-functional source code.

We further developed a Rust compiler plugin to determine whether the functions and blocks in the vulnerability-fix commits marked `unsafe`. First, we extracted the two versions of affected files before and after the fix commit. Second, we extracted the line numbers of vulnerable and fixing code for each affected file u (i.e., compilation unit) before and after the commit, namely, $LineVul^u$ and $LineFix^u$. We denote the code ranges of a function and an unsafe block in the compilation unit u by $FRange$ and $UBRange$, respectively. Third, the plugin identified the code ranges of each function in the affected files before and after the commit, namely, $(FRange, FRange^{'})$, as well as the code ranges of each `unsafe` block, namely, $(UBRange, UBRange^{'})$. Finally, to locate vulnerable code with respect to functions and unsafe blocks in a compilation unit, we checked whether the range of vulnerable code is inside $FRange$ and $UBRange$. Specifically, for each function f in a compilation unit u, we checked whether there exists $LineVul^u$ in the corresponding affected file, such that $LineVul^u \cap FRange_f \neq \emptyset$. If so, we considered the function f as a vulnerable function. Similarly, we identified vulnerable `unsafe` block b, such that $LineVul^u \cap UBRange_b \neq \emptyset$. Likewise, to locate fixing code with respect to functions and unsafe blocks in a compilation unit, the plugin checked whether the range of fixing code is inside $FRange'$ and $UBRange'$. Specifically, for each function f in a compilation unit u, we checked whether there exists $LineFix^u$ in the corresponding affected file, such that $LineFix^u \cap FRange^{'}_f \neq \emptyset$. If so, we considered the function f as a fixing function. Similarly, we identified fixing `unsafe` block b, such that $LineFix^u \cap UBRange^{'}_b \neq \emptyset$.

3.3 Characterizing Vulnerabilities, Vulnerable Packages, and Fixes

Vulnerability categorization (RQ1). The vulnerabilities from the four sources use two classification schemes, i.e., Common Weakness Enumeration (CWE) and RustSec categorization. To categorize disclosed vulnerabilities in the Rust ecosystem, we leveraged Software Fault Patterns (SFP) [31] to build connections between two classification schemes of vulnerabilities. As shown in Table 1, we identified 17 vulnerability types in our dataset (the *Vulnerability Type* column), with the corresponding CWE IDs and RustSec categorization in the *CWE ID* and *RustSec Category* columns. We further categorized the disclosed vulnerabilities in our dataset into the 17 vulnerability types.

Vulnerability life spans (RQ1). Upon a vulnerability's disclosure, we might ask how long it plagued a code repository before a developer fix the vulnerability. We denote the duration as the *life span* of the vulnerability in the code repository, which is investigated in prior work [27]. In the life span of a vulnerability, we also measured two duration: (1) the duration between the introduction and disclosure of a vulnerability, which reflects the window of opportunity for attackers who silently discover a vulnerability to leverage it offensively, before any defensive measures are taken, and (2) the duration between the disclosure and fixing of a vulnerability, which affects the the remediation process and the potential impact of the vulnerability. Reliably determining when a vulnerability was born and fixed automatically is challenging, as it requires understanding the source code and the nature of the vulnerability [27]. Thus, we utilized the collected and processed vulnerability-fix commits. Specifically, for all lines of code deleted by a fix commit, i.e. $LineVul$, we used *git blame* to retrieve the last modification date of each line [50]. Note that we ignore the commits with only additions due to newly added lines did not exist prior to the commit. We

Table 1. Vulnerability types with mappings between classification schemes.

Vulnerability Type	CWE ID	RustSec Category
Memory Access	118, 119, 120, 121, 122, 125, 126, 127, 131, 135, 170, 416, 467, 476, 588, 785, 787, 824	memory-exposure
Memory Management	415, 590, 761, 762, 763	memory-corruption
Synchronization	362, 363, 364, 366, 367, 370, 412, 413, 414, 543, 567, 585, 609, 638, 662, 667, 764, 765	thread-safety
Tainted Input	15, 20, 74, 77, 78, 643, 644, 652, 687, 129	format-injection
Resource Management	400, 404, 459, 672, 674, 770, 774, 772, 789	denial-of-service
Exception Management	248, 252, 253, 273, 280, 390, 431, 478, 484, 584, 600, 665, 908, 909	-
Cryptography	327, 347, 1240	cryptography
Risky Values	28, 190, 194, 369, 456, 466, 468, 475, 480, 486, 562, 570, 579, 587, 594, 597, 681, 685, 704, 768, 843	-
Path Resolution	22, 30, 42, 51, 57, 58, 59, 62, 64, 65, 67, 73, 243, 706	file-disclosure
Information Leak	8, 14, 117, 200, 214, 226, 244, 256, 311, 374, 403, 495, 501, 523, 532, 591, 598, 607, 642, 668, 767	-
Privilege	269, 272	privilege-escalation
Predictability	330, 338, 340	-
Authentication	259, 293, 306, 307, 321, 350, 360, 422, 425, 565, 605, 620, 295	-
API	111,242,245,382,474,477,479,558,572,586,589,617,676,758	-
Access Control	279, 285, 424	-
Failure to Release Memory	401	-
Other	188, 193, 657, 670, 682, 697, 835	code-execution

conservatively designate the earliest blame date across all lines as the estimated date of vulnerability introduction. We used the commit date of the fix commit to estimate when the vulnerability is fixed. In case the vulnerability had multiple fix commits, we conservatively designated the most recent commit date of the fix commits as the estimated date of vulnerability fix.

Vulnerability evolution (RQ1). We investigated the evolution of numbers of vulnerabilities that are introduced and disclosed in the Rust ecosystem over time. To mitigate the impact from the package growth on vulnerability evolution, we normalized the number of disclosed vulnerabilities to both the number of packages and the lines of code (LOC) of vulnerable packages in the ecosystem. In addition, we compared the evolution of numbers of disclosed vulnerabilities across vulnerability types and package categories.

Affected versions of vulnerable packages (RQ2). We extracted package names from the vulnerability reports and identified the unique set of packages that contained at least one vulnerability. For each vulnerable package, we aggregated the affected versions by each of its corresponding vulnerabilities as indicated in the vulnerability reports. We also analyzed the packages with the most vulnerabilities to investigate whether the package popularity would impact the number of vulnerabilities disclosed in a package.

Vulnerable package categorization (RQ2). We categorized the vulnerable packages by referring to the categorization information provided by `crates.io`. Given `crates.io` does not provide categorization information for 165 vulnerable packages, we identified an average of 2.10 package categories (median = 2) for the rest 172 vulnerable packages with categorization information. We further compared the numbers of vulnerabilities, total packages and downloads, as well as the distributions of vulnerability types across different package categories.

Vulnerable code locality (RQ2). For each vulnerability, we first counted the numbers of files, functions, and unsafe blocks that are touched by the corresponding vulnerable code. We then used the total numbers of functions and unsafe blocks in affected versions of vulnerable packages (i.e.,

Table 2. Descriptive statistics of vulnerable packages.

	# Safe Functions	# Unsafe Functions	# Unsafe Blocks
mean (μ)	637.22	16.69	77.30
median (M)	226	2	14
min	2	0	0
max	7,804	361	1,804
std	1,176.03	49.07	181.58
total	137,003	3,589	16,620

the version before vulnerability fixes) as the baseline for normalization. For code that failed in compilation, we used regular expression to estimate the numbers of functions and unsafe blocks. The resulting baseline is shown in Table 2: the vulnerable packages contain an average of 637.22 safe functions, 16.69 unsafe functions and 77.30 unsafe blocks (a median of 226 safe functions, 2 unsafe functions, and 14 unsafe blocks). With the baseline, we further measured the ratios of unsafe functions and unsafe blocks touched by vulnerable code in vulnerable packages, and compared them with the corresponding ratios in the complete code of vulnerable packages. In addition, we compared the vulnerable code localities across different vulnerability types in terms of numbers of commits, files, safe and unsafe functions, and unsafe blocks.

Fix commit complexity and locality (RQ3). To investigate the complexity of a fix commit, we used lines of code (LOC) touched by the fix commit, .i.e, its vulnerable and fixing code, as a simple-albeit-rudimentary metric as with prior studies [24, 32, 50, 67]. Meanwhile, to investigate the locality of a fix commit, we first counted the numbers of functions, unsafe functions, and unsafe blocks touched by its vulnerable and fixing code. In addition, we compared the localities of vulnerability fix commits across vulnerability types in terms of numbers of commits, files, safe and unsafe functions, and unsafe blocks.

Fix patterns (RQ3). We inspected vulnerability fix commits and summarized fix patterns in the fix commits with identical locality category. Each fix commit could fall into multiple locality categories from three categories: (1) the *safe function* category, if the vulnerable code in the fix commit includes safe function(s), (2) the *unsafe function* category, if its vulnerable code includes unsafe function(s), and (3) the *unsafe block* category, if its vulnerable code includes unsafe block(s).

4 RESULTS

In this section, we present the results of our research questions that investigate the security risks in the Rust ecosystem.

4.1 RQ1: Vulnerabilities in the Rust Ecosystem

We investigated the characteristics of disclosed vulnerabilities in the Rust ecosystem, including the vulnerability types, life spans, and the evolution of the number of vulnerabilities.

Types of vulnerabilities. We collected a total of 433 unique vulnerabilities in the Rust ecosystem, out of which 73 have not been categorized, leaving 360 vulnerabilities that are categorized with a median of 1 vulnerability type (min: 1, max: 4, mean: 1.65, std: 0.76). Table 3 presents the overall distribution of vulnerabilities across 17 vulnerability types. Memory safety and concurrency issues account for 63.6% of the 360 categorized vulnerabilities.

Memory safety issues involve *memory access* (39.17%) and *memory management* (40.00%) vulnerability types, accounting for 59.7% of the categorized vulnerabilities. The *memory access* vulnerabilities usually arise from buffer or pointer access problems, e.g., *buffer overflow*, *use after free*, and *null pointer deference*. We take the RUSTSEC-2021-0128 in the `rusqlite` package as an

Table 3. Distribution of vulnerabilities across vulnerability types. "Percentage" denotes the number of vulnerabilities belonging to a specific vulnerability type divided by the number of categorized vulnerabilities, .i.e, count/360. "Disclosure Duration" denotes the median duration between introduction and disclosure reported in days. "Fix Duration" denotes the median duration between disclosure and fix reported in days.

Vulnerability Type	Count	(with Fix)	Percentage	Disclosure Duration	Fix Duration
Memory Management	144	(67)	40.00%	668.0	1.0
Memory Access	141	(53)	39.17%	678.0	0.0
Synchronization	74	(44)	20.56%	770.5	0.0
Tainted Input	46	(23)	12.78%	780.0	-5.0
Resource Management	40	(37)	11.11%	599.0	-2.0
Exception Management	38	(18)	10.56%	1062.5	18.0
Cryptography	26	(8)	7.22%	757.5	-2.0
Other	26	(10)	7.22%	419.5	-1.0
Risky Values	21	(10)	5.83%	802.0	-2.0
Path Resolution	14	(12)	3.89%	165.5	-2.0
Information Leak	9	(3)	2.50%	80.0	-14.0
Privilege	4	(1)	1.11%	76.0	-5.0
Predictability	3	(2)	0.83%	107.5	-2.0
Authentication	3	(0)	0.83%	/	/
API	2	(1)	0.56%	846	30.0
Access Control	2	(0)	0.56%	/	/
Failure to Release Memory	1	(0)	0.28%	/	/

example of *memory access* vulnerability. In the vulnerable code of the `rusqlite` package affected by RUSTSEC-2021-0128, the lifetime bounds on several closure-accepting functions are so loose that allow the access to dropped objects on the stack thus cause *use after free* error. The *memory management* vulnerabilities are due to problems in memory allocation or deallocation, e.g., *double free*. RUSTSEC-2021-0033 in the `stack_dst` package is an example of *memory management* vulnerabilities. Specifically, the `push_cloned` function in the vulnerable code of `stack_dst` package deallocates uninitialized memory thus cause *double free* error.

Concurrency issues involve *synchronization* vulnerabilities, which rank the third in the frequency of occurrence across different types of vulnerabilities (20.56%). The *Synchronization* vulnerabilities occur when multiple processes or threads share resources, including race condition and misuse of locks. For instance, the unsafe `Send` trait implementation in the `atom` package involved in RUSTSEC-2020-0044 causes data race error.

Vulnerability life spans. Figure 8 illustrates the distribution of disclosure duration for vulnerabilities. It takes an average of 770 days (2.1 years) for a vulnerability to be disclosed after the vulnerability was introduced in a Rust package (median: 693, min: 2, max: 2,364, std: 534.2). As the per-vulnerability median indicates, 50% of the vulnerabilities had disclosure duration exceeding 693 days (1.9 years). Our observations concur with prior findings that vulnerabilities in the npm ecosystem are disclosed within a median of 24 months, considerably shorter than the 37 months required for the vulnerabilities in the PyPI ecosystem [1]. The vulnerability with the longest disclosure duration (2,364 days) is RUSTSEC-2022-0029 in the *crossbeam* package, introduced in December 2015 and disclosed in June 2022. 1.8% of the vulnerabilities had a disclosure duration lower bound of less than 30 days, most of which are introduced after August 2021, indicating an increase in the security awareness within the Rust ecosystem.

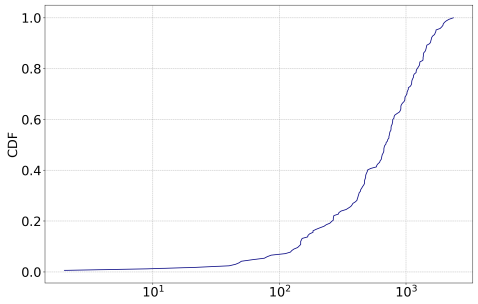


Fig. 8. CDFs of the duration between the introduction and disclosure of a vulnerability.

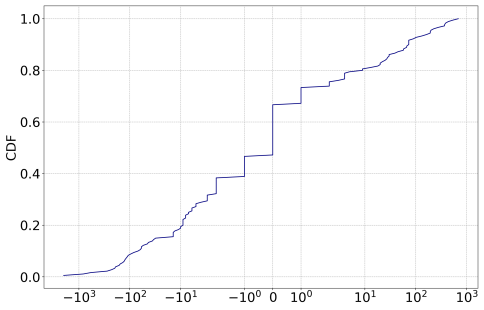


Fig. 9. CDFs of the duration between the disclosure and fixing of a vulnerability.

In Figure 9, we depict the cumulative distribution function (CDF) of the number of days between disclosure and fixing. The predominant behavior in Figure 9, observed for two-thirds of the vulnerabilities (120 out of 180), is that the vulnerability fixes were committed by disclosure time, manifesting as negative or zero time differences. The predominant behavior suggests that the majority of vulnerabilities in the Rust ecosystem were either internally discovered or disclosed to project developers using private channels, which is the expected best practice for vulnerability disclosure [70]. In Figure 9, vulnerabilities disclosed but not yet fixed manifest as positive time difference values, which occurred for one-third of vulnerabilities (60 out of 180) in the Rust ecosystem. The percentage of unpatched vulnerabilities by disclosure is higher than the 21.2% as reported in prior study [27] and comparable to the 30% for Windows vulnerabilities [20]. The 60 vulnerabilities with positive time difference values have an average of 88 days of their fixing duration (median: 23.5, min: 1, max: 686, std: 146.0). The vulnerability with the longest fix duration (686 days) was RUSTSEC-2017-0006 in the *rmpv* package, which was disclosed in November 2017 and fixed in October 2019. In addition, approximately 42% of the vulnerabilities remain unpatched for more than 30 days after their disclosure, leaving a window of opportunity for potential attacker exploitation.

We further compared the duration of vulnerability disclosure and fixing across vulnerability types as shown in Table 3. We observe that the duration of vulnerability disclosure and fixing vary widely across vulnerability types. Generally, frequently occurred vulnerability types (Count > 20) tend to have a significantly longer duration of disclosure compared to rarely occurred vulnerability types (Count < 20), as supported by Wilcoxon rank-sum tests (p-value = 0.0266). Among the rarely occurred vulnerability types, the *API* vulnerabilities turn out to be an exception with the longest duration of disclosure and fixing across vulnerability types.

Vulnerability evolution. We present the evolution of the number of vulnerabilities disclosed over time in the Rust ecosystem in Figure 10. We observe that the number of vulnerabilities disclosed grows slowly from November 2014 to November 2020, and experiences two rapid growth phases. The first rapid growth starts from November 2020 and ends in March 2021, while the second occurs in July 2021. The first rapid growth may attribute to a large-scale campaign during that period of time, in which RustSec has published 129 memory safety vulnerabilities as part of the research efforts made by Bae et al. [4]. During the second rapid growth, the number of disclosed vulnerabilities increased from 197 to 330 due to unrestricted Send or Sync on generic types, which is also discussed in prior work [4]. The evolution in the numbers of vulnerabilities vary widely across vulnerabilities types as shown in Figure 11. The *memory access*, *memory management* and *synchronization* vulnerabilities grow fastest over time, with a growth pattern that resembles the evolution of vulnerabilities disclosed in the Rust ecosystem as shown in Figure 10.

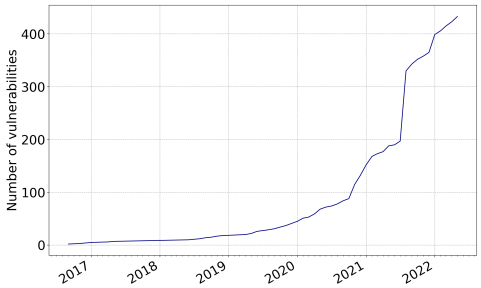


Fig. 10. Evolution of number of vulnerabilities disclosed over time.

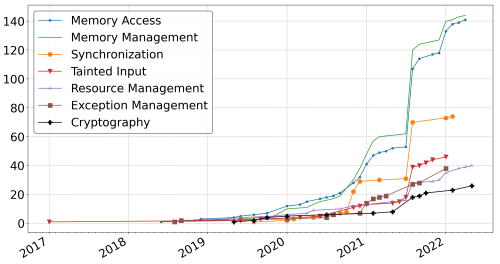


Fig. 11. Evolution of numbers of disclosed vulnerabilities across vulnerability types.

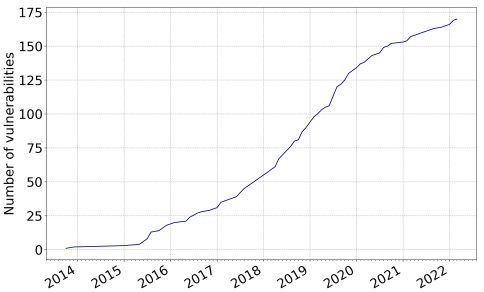


Fig. 12. Evolution of number of vulnerabilities introduced into Rust code repositories over time.

We also present the evolution of the number of vulnerabilities introduced into Rust code repositories over time in Figure 12, which demonstrates a linear growth rate from July 2015 to January 2020 (R^2 = 0.9561) and becomes stabilized after March 2020. The numbers of vulnerabilities disclosed and introduced demonstrate different growth rates over time, which may be due to the increase of individuals and organizations participated in Rust vulnerability discovery as Rust becomes increasingly popular in systems software development. Another possible reason could be the development and application of vulnerability detection tools in the Rust ecosystem, which facilitate the discovery of vulnerabilities [4, 28].

Next, we investigated whether package growth in the Rust ecosystem contributes to the increase of disclosed vulnerabilities. As shown in Figure 13, the normalized number of vulnerabilities disclosed per 1,000 packages grows from one in 2017 to five in 2022, indicating an increase of package-wise security risks in the Rust ecosystem. Meanwhile, the normalized number of vulnerabilities disclosed per 100,000 lines of code in vulnerable packages reaches the peak (2.1) in August 2020 after three climbing stages, and have experienced a sharp decrease to 0.5 since then, as shown in Figure 14, suggesting a decreasing tendency in security risks per lines of code in the Rust ecosystem.

Finally, we investigated how vulnerabilities with different types evolve over time with respect to package category as shown in Figure 15. We make several observations:

- *Memory management* vulnerabilities are disclosed across package categories, but with different frequencies of occurrence and growth rates.

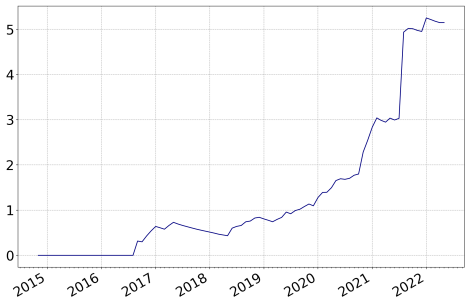


Fig. 13. Disclosed vulnerabilities per 1,000 packages.

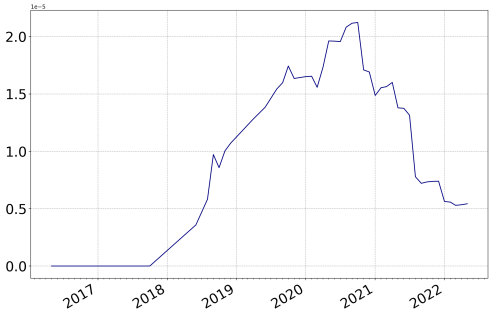


Fig. 14. Disclosed vulnerabilities per 100,000 lines of code in vulnerable packages.

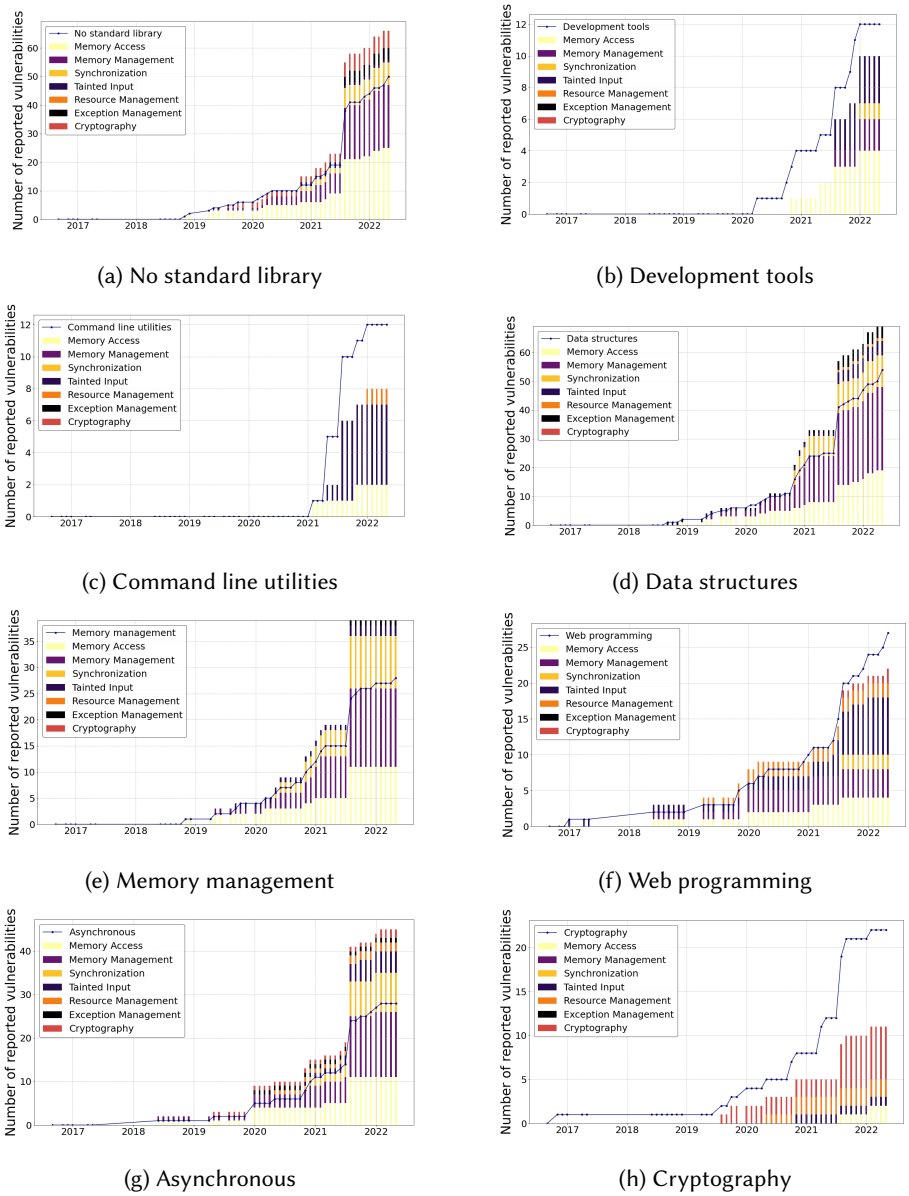
- *Command line utilities* packages have no vulnerability disclosed until April 2021, and have 12 vulnerabilities disclosed in total until April 2022, indicating relatively low security risk over time (Figure 15c).
- *No standard library* and *data structures* packages both have more than 60 vulnerabilities disclosed in total until April 2022, among which *memory management* vulnerabilities account for a majority (44.0% in *no standard library* and 53.7% in *data structures*) (Figure 15a and Figure 15d). The numbers of disclosed vulnerabilities are greater than any other package categories, indicating relatively higher security risk over time.

Summary for RQ1: The top three vulnerability types in the Rust ecosystem are *memory access*, *memory management*, and *synchronization*, accounting for 63.6% of categorized vulnerabilities and exhibiting the fastest growth rates. It takes over 2 years for the vulnerabilities to be publicly disclosed, among which 66.7% have fixes committed before their disclosure. The number of disclosed vulnerabilities experiences two rapid growth periods, while the number of vulnerabilities introduced into code repositories grows linearly. Normalized numbers of disclosed vulnerabilities suggest a continuously increasing trend in package-level security risks over time, yet a decreasing trend in code-level security risks since August 2020. In addition, the security risks in the Rust ecosystem vary widely across different package categories.

4.2 RQ2: Vulnerable Packages in the Rust Ecosystem

We identified a total of 337 vulnerable packages, accounting for 0.40% of packages in the Rust ecosystem. 120 out of 337 vulnerable packages remain unpatched. The 337 vulnerable packages have an average of 1.3 disclosed vulnerabilities (min: 1, max: 14, median: 1, std: 1.04). The disclosed vulnerabilities affect an average of 28.6 versions of the Rust packages (min: 1, max: 339, median: 17, std: 34.33), accounting for 75.09% of the versions per package on average (min: 1.37%, max: 100%, median: 82.07%, std: 0.27).

Popularity of vulnerable packages. On the one hand, popular packages tend to have more vulnerabilities. Specifically, the top 5 Rust packages with the most vulnerabilities are (1) `openssl-src`, with 14 vulnerabilities and 55 versions affected (87.27%), (2) `wasmtime`, with 7 vulnerabilities and 62 versions affected (58.06%), (3) `hyper`, with 7 vulnerabilities and 224 versions affected (92.41%), (4) `ckb`, with 7 vulnerabilities and 32 versions affected (28.13%), and (5) `smallvec`, with 5 vulnerabilities and 54 versions affected (68.52%). Among the top 5 packages, `openssl-src`, `hyper` and `smallvec` have over 9 millions of downloads till April 2023. One possible reason could be that



(a) No standard library

(b) Development tools

(c) Command line utilities

(d) Data structures

(e) Memory management

(f) Web programming

(g) Asynchronous

(h) Cryptography

Fig. 15. Evolution of numbers of disclosed vulnerabilities across package categories.

popular packages have far more developers and users than less popular ones. The larger community of developers and users for popular Rust packages tend to uncover more vulnerabilities compared to less popular packages. On the other hand, some popular packages officially published by September 2016 have no vulnerabilities disclosed, e.g., `libc` and `syn`, which have over 100 million downloads. In addition, we observe that unpopular packages (with less than 100,000 downloads) tend to suffer from vulnerabilities for more versions. Particularly, among the 109 vulnerable packages with all

Table 4. Characteristics of package categories with vulnerable packages.

Package Category	# Vulnerabilities	# Packages	# Downloads	# Vulnerabilities per Package
Memory management	28	468	397,961,797	5.98%
Concurrency	26	909	799,571,861	2.86%
Data structures	54	2,144	1,170,205,196	2.52%
Caching	5	234	68,948,180	2.14%
Network programming	33	2,138	816,412,017	1.54%
Asynchronous	28	1,900	1,039,591,569	1.47%
Encoding	22	1,494	904,623,837	1.47%
No standard library	50	3,671	3,704,866,624	1.36%
Rust patterns	15	1,158	829,299,334	1.30%
Parsing tools	8	1,272	363,898,651	1.26%
Text processing	11	943	484,155,024	1.17%
Cryptography	22	1,933	912,432,279	1.14%
Web programming	27	2,386	650,382,813	1.13%
Algorithms	17	1,595	1,190,075,755	1.07%
Operating systems	8	1,013	603,471,887	0.79%
API bindings	10	2,265	417,282,873	0.44%
Development tools	12	3,725	1,640,269,851	0.32%
Command line utilities	12	4,418	25,115,904	0.27%
(vulnerabilities<=6)	90	14,670	2,461,533,553	0.61%
Non-categorized	214	55,895	-	0.38%

versions affected by vulnerabilities, 76 packages have less than 100,000 downloads in total over time.

Categories of vulnerable packages. Table 4 reports the numbers of disclosed vulnerabilities, packages, and downloads till May 2022, as well as the average numbers of vulnerabilities per package across package categories. The top 3 package categories with the most vulnerabilities are *data structures*, *no standard library* and *network programming*. In the meantime, *memory management*, *concurrency* and *data structures* rank the top 3 among package categories in terms of the average number of vulnerabilities per package. Interestingly, the *memory management* category has fewer packages (468) and downloads (around 400 million downloads), but relatively more vulnerabilities disclosed (28), compared with other package categories, indicating that *memory management* packages are more prone to vulnerabilities.

We further compare the distributions of vulnerability types across package categories. As shown in Table 5, we chose the union of the top 5 categories with vulnerable package percentage and the top 5 categories with vulnerabilities in Table 4 for analysis. The distributions of vulnerability types vary substantially across package categories: The *memory management* package category tends to have more *memory access*, *memory management* and *synchronization* vulnerabilities; the *concurrency* packages tend to have more *synchronization* and *memory management* vulnerabilities; and the *data structure* packages tend to have more *tainted input* vulnerabilities. To see if the differences in the distributions of vulnerability types are statistically significant, we conduct Wilcoxon signed-rank tests [63] with Bonferroni correction at 95% significance level. As a result, we observe statistically significant differences (1) *caching* vs. *data structures* (p-value = 0.0490) and (2) *caching* vs. *no standard library* (p-value = 0.0490), indicating the penitential impact of package categories on disclosed vulnerability types in the Rust ecosystem.

Vulnerability locality in vulnerable packages. As illustrated in Table 6, a disclosed vulnerability affects 1.85 files, 3.35 safe functions, 0.15 unsafe functions, and 1.39 unsafe blocks on average in

Table 5. Distributions of vulnerability types in top 5 package categories with vulnerabilities and top 5 package categories with vulnerable package percentages.

Vulnerability Type	Memory Management	Concurrency	Data Structures	Caching	No Standard Library	Network Programming
Memory Access	11	8	19	2	25	11
Memory Management	15	17	29	3	22	10
Synchronization	10	17	11	4	8	4
Tainted Input	2	4	5	0	1	3
Resource Management	0	1	1	0	0	4
Exception Management	1	0	4	0	4	3
Cryptography	0	0	0	0	6	3
Other	0	0	3	0	4	2
Risky Values	2	0	4	0	3	1
Path Resolution	1	0	1	0	0	3
Predictability	0	0	0	0	2	0
API	0	0	0	0	1	0
Failure to Release Memory	0	0	1	0	0	0
Total	28	26	54	5	50	33

Table 6. Descriptive statistics of vulnerability locality in vulnerable packages.

	# Files	# Safe Functions	# Unsafe Functions	# Unsafe Blocks
mean (μ)	1.85	3.35	0.15	1.39
median (M)	1	1	0	0
min	1	0	0	0
max	14	83	4	50
std	1.95	8.84	0.53	5.20
total	395	684	31	284

Fig. 16. Ratios of unsafe functions in vulnerable code vs. all code of vulnerable packages.

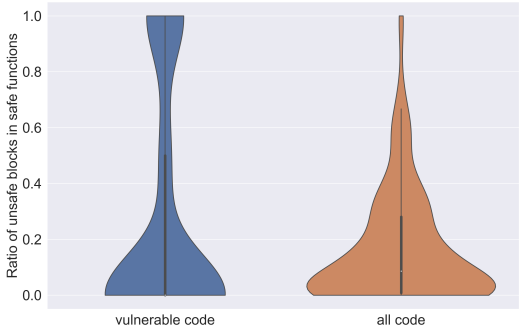


Fig. 17. Ratios of unsafe blocks in vulnerable code vs. all code of vulnerable packages.

vulnerable packages in the Rust ecosystem (1 file, 1 safe function, 0 unsafe function, and 0 unsafe block in median). The small median number of affected files per fix commit indicates that vulnerable code in the Rust ecosystem is localized at the file level. At functional level, 95% of the affected

Table 7. Descriptive statistics of vulnerability locality across vulnerability types.

Vulnerability type		Memory Access	Memory Management	Synchronization	Tainted Input	Resource Management	Exception Management	Path Resolution
# Commits		42	28	39	21	9	6	6
# Files	μ	2.30	2.14	1.59	1.57	1.89	2.33	1.83
	M	1	1	1	1	2	2	1.5
	min	1	1	1	1	1	1	1
	max	14	14	5	4	3	5	3
	std	2.94	2.72	1.25	0.95	0.93	1.51	0.98
# Safe functions	μ	3.88	4.32	1.05	1.43	2	8.67	1.33
	M	1	1	0	1	2	3	1
	min	0	0	0	0	1	1	1
	max	59	33	13	5	5	40	3
	std	10.92	8.56	2.67	1.12	1.32	15.41	0.82
# Unsafe functions	μ	0.17	0.14	0.15	0.05	0	0	0
	M	0	0	0	0	0	0	0
	min	0	0	0	0	0	0	0
	max	2	3	2	1	0	0	0
	std	0.44	0.59	0.49	0.22	0	0	0
# Unsafe blocks	μ	2.14	2.39	0.21	0.19	0	7.67	0
	M	0	0	0	0	0	2	0
	min	0	0	0	0	0	0	0
	max	50	67	6	3	0	40	0
	std	7.85	6.32	0.98	0.68	0	15.87	0

functions are safe functions; Among the affected safe functions, 41.5% contain unsafe blocks in their body of a function. One possible reason for the high percentage of safe functions in vulnerable code is that developers tend to wrap unsafe code in safe functions and provide conditional checks in those functions before entering the unsafe code, which is in line with the Rust idiomatic style to encapsulate unsafety [4].

We further compared the ratios of unsafe functions and unsafe blocks between vulnerable code vs. all code in vulnerable packages as shown in Figure 16 and Figure 17, respectively. We observe that vulnerable packages have higher ratios of unsafe functions (0.059 vs. 0.034 on average) and unsafe blocks (0.261 vs. 0.165 on average) in their vulnerable code as compared to their complete code. The Wilcoxon signed-rank test at 95% significance level suggests statistically significant differences exist in the ratios of unsafe functions (p-value = 0.002) and unsafe blocks (p-value = 0.017) between vulnerable code and complete code in vulnerable packages.

Finally, we compared the vulnerability localities across vulnerability types, including the numbers of commits, files, safe and unsafe functions, and unsafe blocks, as shown in Table 7. We make the following observations:

- The *exception management* vulnerabilities tend to be the least localized at the file and function levels, considering the greatest numbers of files and safe functions affected by them as compared to other vulnerability types.
- The *synchronization* vulnerabilities tend to be the most localized at the function level, considering the smallest number of safe functions on average they affected.
- The *memory management* and *memory access* vulnerabilities show the tendency to locate more frequently in safe functions than in unsafe functions, considering they affected more safe functions than unsafe ones.
- The *resource management* and *path resolution* vulnerabilities originate from the safe code in safe functions, considering they affected neither unsafe functions nor unsafe blocks.

Table 8. Descriptive statistics of vulnerability fixes.

	# Lines Added	# Lines Deleted	# Safe Functions	# Unsafe Functions	# Unsafe Blocks
mean (μ)	41.13	18.17	3.85	0.16	1.49
median (M)	14	4	1	0	0
min	1	0	0	0	0
max	665	330	83	4	50
std	81.97	39.75	9.12	0.53	5.24
total	-	-	786	32	304

Summary for RQ2: Vulnerable packages in the Rust ecosystem have an average of 1.3 disclosed vulnerabilities that affected 28.6 versions of the corresponding packages. Popular packages tend to have more vulnerabilities, while less popular ones tend to suffer from vulnerabilities for more versions. *Memory management* is the most vulnerable package category, with a small number of packages but a large number of vulnerabilities. In addition, the *memory management* package category tends to have more *memory access*, *memory management*, and *synchronization* vulnerabilities as compared to other package categories. The *exception management* vulnerabilities tend to be the least localized at the file and function levels as compared to other vulnerability types. In vulnerable packages, the vulnerable code tends to involve statistically significantly more unsafe functions and unsafe blocks as compared to complete code, which is localized at the file level.

4.3 RQ3: Vulnerability Fixes in the Rust Ecosystem

Fix commit complexity and locality. As shown in Table 8, the commits of vulnerability fixes in the Rust ecosystem involve an average of 41 and 18 LOC added and deleted, respectively (14 and 4 LOC added and deleted in median). The commits of vulnerability fixes have touched 3.85 safe functions, 0.16 unsafe functions, and 1.49 unsafe blocks on average. Safe functions account for 96% of all the functions touched by the vulnerability fixes, among which 38.8% contain unsafe blocks.

Next, we compared the locality of vulnerability fix commits across vulnerability types with respect to the numbers of safe functions, unsafe functions, and unsafe blocks touched by the commits as shown in Table 9. The fix commits of *exception management* vulnerabilities involve the most safe functions compared to other vulnerability types (8.67), indicating the widest spread of touched code and potential challenges when fixing *exception management* vulnerabilities in practice. On the contrary, the fix commits of *synchronization* vulnerabilities touch the fewest safe functions compared to other vulnerability types (1.21), indicating the most localized fixes across vulnerability types.

Vulnerability fix patterns. To capture how developers fix different types of vulnerabilities, we first compared the numbers of safe functions, unsafe functions, and unsafe blocks affected by vulnerabilities (Table 7) with those touched by corresponding fixes (Table 9). We make the following observations: (1) The *tainted input* vulnerabilities demonstrate the greatest increase in the number of unsafe blocks compared to other vulnerability types, from 0.19 to 0.24; The increase indicates that developers tend to add unsafe blocks when fixing *tainted input* vulnerabilities. (2) The *path resolution* vulnerabilities have the greatest increase in the number of safe functions, from 1.33 to 2.17, indicating the addition of safe functions when fixing such vulnerabilities. (3) The *path resolution* vulnerabilities have the greatest increase in the number of unsafe functions, from 0.17 to 0.19, indicating the addition of unsafe functions when fixing such vulnerabilities.

Table 9. Statistics of commit locality of vulnerability fixes across vulnerability types.

Vulnerability Type		Memory Access	Memory Management	Synchronization	Tainted Input	Resource Management	Exception Management	Path Resolution
# Safe Functions	μ	4.0	4.71	1.21	2.0	3.44	8.67	2.17
	M	1	2	0	1	2	3	2
	min	0	0	0	0	1	1	1
	max	59	33	16	8	12	40	5
	std	10.89	8.60	3.22	1.79	3.50	15.41	1.47
# Unsafe Functions	μ	0.19	0.14	0.15	0.05	0	0	0
	M	0	0	0	0	0	0	0
	min	0	0	0	0	0	0	0
	max	2	3	2	1	0	0	0
	std	0.45	0.59	0.49	0.22	0	0	0
# Unsafe Blocks	μ	2.17	2.39	0.21	0.24	0	7.67	0
	M	0	0	0	0	0	2	0
	min	0	0	0	0	0	0	0
	max	50	33	6	3	0	40	0
	std	7.84	6.32	0.98	0.70	0	15.87	0

Table 10. The characteristics of vulnerability fix commits across localities.

Vulnerability Locality	Fix Pattern	Count
Safe function	add safe functions	28
	remove safe functions	12
	modify safe functions	117
Unsafe function	add unsafe functions	0
	remove unsafe functions	0
	modify unsafe functions	16
Unsafe block	add functions or lines in function	5
	remove unsafe blocks	27
	modify unsafe blocks	36

We further summarized the resulting fix patterns of our our manual inspection with respect to different vulnerability localities as shown in Table 10. In general, we observe three operations, i.e., *addition*, *deletion*, and *modification* of code, when developers fix vulnerabilities. Among the three operations, the modification operation accounts for the majority of fix commits across different vulnerability localities. We also make the following observations in particular:

Observation 1: Developers tend to add safe functions, or add lines in safe functions to fix vulnerable safe functions. Developers tend to add safe code when fixing vulnerabilities that locate in safe functions, rather than removing existing code.

The added safe functions or lines validate pre-conditions or customize the default implemented functions, thus fix corresponding vulnerabilities. Taking the fix of the *denial-of-service* vulnerability (CVE-2022-24713) as an example (Listing 3), the fix inserts an additional function `c_empty()` to deal with an empty string, which may cause denial of service.

Listing 3. Example vulnerability fix (CVE-2022-24713).

```
// code snapshot before fix commit
fn c(&mut self, expr: &Hir) -> ResultOrEmpty {
    ...
    match *expr.kind() {
        Empty => Ok(None),
        ...
    }
}

// code snapshot after fix commit
fn c_empty(&mut self) -> ResultOrEmpty {
    self.extra_inst_bytes +=
        std::mem::size_of::<Inst>();
    Ok(None)
}
fn c(&mut self, expr: &Hir) -> ResultOrEmpty {
    ...
    match *expr.kind() {
        Empty => self.c_empty(),
        ...
    }
}
```

Observation 2: Developers tend to remove unsafe blocks to fix vulnerable unsafe blocks. Developers tend to remove existing code, rather than adding code to fix vulnerabilities that locate in unsafe blocks. Taking the fix of the memory exposure vulnerability (RUSTSEC-2021-0086) as an example (Listing 4), the fix removes the unsafe block and zeroes out the buffer `buf` before further operations can be undertaken.

Listing 4. Example vulnerability fix (RUSTSEC-2021-0086).

```
// code snapshot before fix commit
let mut buf = Vec::with_capacity(frame.data_size);
unsafe { buf.set_len(frame.data_size) };

// code snapshot after fix commit
let mut buf = vec![0; frame.data_size];
```

Observation 3: Developers tend to modify unsafe trait implementations to fix vulnerable unsafe functions. Specifically, we identified a *data race* pattern that occurs in disclosed vulnerabilities that locate in unsafe functions as shown in Listing 5. In such vulnerability pattern, objects do not restrict to sendable or syncable types when they implement `Send` or `Sync` traits, leading to the sharing of non-syncable types across threads in concurrent programs. To fix such vulnerabilities, developers tend to use `Send` or `Sync` traits as bounds to stipulate the functionality that the generic type `T` must implement.

Listing 5. Vulnerability fix pattern of *data race* vulnerabilities.

```
// code snapshot before fix commit
unsafe impl<T> Send for MyObject<T> {}
unsafe impl<T> Sync for MyObject<T> {}

// code snapshot after fix commit
unsafe impl<T: Send> Send for MyObject<T> {}
unsafe impl<T: Sync> Sync for MyObject<T> {}
```

Summary for RQ3: The vulnerability fix commits in the Rust ecosystem involve a median of 14 lines of code added, and a median of 4 lines of code deleted, suggesting that vulnerability fix commits are typically localized. 96% of the functions touched by these fix commits are safe functions. The fix commits of *exception management* vulnerabilities involve the highest proportion of safe functions compared to other vulnerability types, indicating potential challenges of fixing *exception management* vulnerabilities in practice. In addition, developers tend to (1) add safe functions or add lines in safe functions to fix vulnerable safe functions, (2) remove unsafe blocks to fix vulnerable unsafe blocks, and (3) modify unsafe trait implementations to fix vulnerable unsafe functions.

5 DISCUSSION

We now summarize our main results, discuss their implications, and highlight the avenues for future research.

Rust is an active and growing software ecosystem at its early stage, coupled with an increasing awareness of risks of security vulnerabilities. The Rust ecosystem hosts over 100 thousand packages on `crates.io` by March 2023, and has been experiencing exponential growth in number of packages and downloads from 2014 to 2022 as observed in our preliminary investigation (Section 2.3). As compared to other ecosystems like npm, PyPI and Maven, which host over 2 million packages[6], 440 thousand packages[7], and 32 million artifacts[8], respectively by March 2023, we observed a significantly smaller number of packages in the Rust ecosystem, indicating Rust is at its early age of development. In addition, the exponential growth in number of packages in the Rust ecosystem resembles the trends of package growth at the early ages of the PyPI [8] and Maven ecosystems [45], suggesting Rust to be an active and growing ecosystem.

Interestingly, our preliminary investigation observed a sharp decline in the increasing number of Rust packages since mid-late 2020, which may be caused by the Mozilla lay-off in (August 2020)[9]. The Rust packages in the *no standard library* category, ranked the second in terms of the total number across package categories, received the most downloads over time. Future work could systematically investigate the factors that influence the evolution of the number of packages, and analyze how the sub-categories in the *no standard library* category could affect the evolution of package downloads in the Rust ecosystem.

The number of vulnerabilities disclosed per 1,000 packages in the Rust ecosystem grows from one in 2017 to five in 2022 (RQ1), indicating an increasing awareness of the risks of security vulnerabilities in the ecosystem. The increasing trends in the number of disclosed vulnerabilities are observed in the PyPI and npm ecosystems as well [8], which may attribute to the coordinated

[6] http://www.modulecounts.com/
[7] https://pypi.org/
[8] https://mvnrepository.com/repos
[9] https://blog.rust-lang.org/2020/08/18/laying-the-foundation-for-rusts-future.html

efforts in the increasing awareness of security risks in the ecosystems and continuous process of testing packages to detect vulnerabilities before exploited.

The majority of the vulnerabilities in the Rust ecosystem relate to memory safety and concurrency issues. Memory safety and concurrency issues account for two-thirds of the vulnerabilities in the Rust ecosystem (RQ1). The frequent occurrence of memory safety and concurrency issues may be due to that developers tend to use the Rust programming language for systems software development, thus manipulating memory and threads in their code frequently. In contrast, cross-site scripting vulnerabilities appear to be the most common type of vulnerability in both PyPI and npm ecosystems as discussed in prior research [8], given Python and JavaScript are popular programming languages for the development of Web applications. We also find that 77% of the vulnerabilities related to memory safety and concurrency issues locate in unsafe code of vulnerable Rust packages (RQ2), suggesting that practitioners should pay more attention to operations related to memory and concurrency when writing unsafe code. For instance, practitioners could enforce bound constraints when implementing `Send` and `Sync` traits to avoid data race as observed in RQ3.

Vulnerabilities in the Rust ecosystem are not localized in unsafe functions, but relate more to unsafe blocks in safe functions as compared to the safe code in safe functions. Safe functions account for 95% of the functions in the vulnerable packages affected by disclosed vulnerabilities (RQ2). The percentage of safe functions in the vulnerable packages affected by disclosed vulnerabilities is close to the percentage of safe functions in the Rust ecosystem as reported in a recent study (95.9%) [36], indicating that vulnerabilities in the Rust ecosystem are not localized in unsafe functions. Meanwhile, 41.5% of the safe functions in the vulnerable packages affected by disclosed vulnerabilities contain unsafe blocks in their function body (RQ2). The frequency of occurrence of unsafe blocks in safe functions in the vulnerable packages is significantly higher than that of all the Rust packages (13.8%) as reported in a recent study [36], indicating that practitioners could put forth more effort on unsafe blocks for securing safe functions compared to their safe code.

Practices towards safer Rust code. 86.67% of the *tainted input* and *resource management* vulnerabilities reside in safe code of vulnerable packages (RQ2), indicating that the *tainted input* and *resource management* vulnerabilities are more localized in safe code than other types of vulnerabilities. The results suggest that practitioners should pay more attention to safe code for data validation when dealing with user input as compared to unsafe code. In terms of fixing vulnerabilities, prior study [27] reported that the median security commit diff involved 7 LOC. The results of RQ3 revealed that the median fix commit diff of the Rust ecosystem includes 14 and 4 LOC added and deleted, respectively, indicating that the fixes in Rust are more complex. Moreover, the study [27] found that 59% of security changes were located in a single function, while the results of RQ3 showed that the median fix commit involves an average of 3.85 safe functions (median = 3.85), suggesting that the fixes are less localized in the Rust ecosystem. In addition, developers tend to modify both safe and unsafe code in vulnerable packages, rather than directly remove unsafe code when fixing vulnerabilities (RQ3), suggesting the necessity of unsafe code in the development of Rust packages, which is in line with the findings of a previous study [19] – practitioners use the unsafe keyword in 29.4% of the Rust libraries. Thus, both safe and unsafe code deserve a comparable consideration when securing Rust packages in practice.

6 THREATS TO VALIDITY

Construct validity. Our dataset contains the complete list of disclosed vulnerabilities in the Rust ecosystem by May 24, 2022, thus it is inevitable that the characteristics of vulnerabilities and fixes in the Rust ecosystem may evolve along with the expansion of vulnerability dataset over time. We analyzed vulnerability fix commits for the investigation of vulnerable packages and fixes in the

Rust ecosystem, which is in line with previous studies [9, 42, 50, 59]. The approach may introduce noise into the dataset given that some code inside the fix commits might not be vulnerable or fix vulnerabilities. To mitigate the threat, we manually inspected the 287 fix commits we have collected and excluded 69 fix commits with irrelevant code, e.g., refactoring and typo fixes. As a result, the vulnerability fix commits in our dataset involve a median of 14 and 4 LOC added and deleted, respectively, whose sizes are comparable to vulnerability fix commits in previous studies [27, 66].
Internal validity. We implemented a Rust compiler plugin and all scripts by ourselves with careful review; despite the extensive testing phase, we may not exclude all possible implementation errors. For the sake of replicability, we made all data and scripts employed publicly available[10]. Some steps in our methodology rely on information produced by the Rust compiler, e.g., identifying the localities of Rust vulnerabilities. Consequently, these steps may be sensitive to unfixed bugs in the Rust compiler.

For the 33 fix commits (15.14%) that fail the compilation process, we used text analysis to collect the numbers of safe/unsafe functions and unsafe blocks in their code. To estimate the potential threat introduced by the text analysis, we used the 185 fix commits that pass the compilation process and compared their numbers of safe/unsafe functions and unsafe blocks identified by text analysis with those identified by the compiler plugin. We found that text analysis identifies 23.7% less unsafe blocks, 10.5% less safe functions, and 14.4% less unsafe functions compared to the compiler plugin. As a result, text analysis would underestimate the total numbers of unsafe blocks, safe functions and unsafe functions by 3.6%, 1.6% and 2.2% in our dataset, respectively. The ratio of unsafe block and unsafe function in all code of vulnerable package would reduce by 2% and 0.6%, respectively. Thus, the slight underestimation of safe/unsafe functions and unsafe blocks would not affect our conclusions in RQ2 on the ratios of unsafe functions and blocks in vulnerable packages.
Conclusion validity. In RQ2, we investigated the affected versions of vulnerable packages. We noticed that some vulnerabilities do not explicitly indicate the earliest affected package versions, and considered the initial releases of their affected packages as the earliest versions affected by those vulnerabilities. The approach is also used in previous studies [68] to estimate the earliest package versions affected by vulnerabilities, which could lead to inaccuracy in the range of affected package versions. To evaluate the inaccuracy of the approach adopted, we took a random sample of 40 vulnerabilities out of 364 that do not specify the earliest affected versions by vulnerabilities, with a 95% confidence interval and 15% sampling error. For each sampled vulnerability, we manually checked the corresponding fix commit(s) to determine whether the vulnerability exists in the initial release of the affected package. We found that 8 out of the sampled 40 vulnerabilities do not exist in the initial releases of the affected packages, leading to an overestimation of affected versions of packages by vulnerabilities. The 95% Agresti-Coull confidence interval is (0.6498, 0.8976).

Memory safety issues account for the majority of disclosed vulnerabilities in the Rust ecosystem. The large amount of *memory related* vulnerabilities may attribute to the fact that security experts and researchers tend to focus on memory problems in Rust packages given that Rust is claimed to ensure memory safety [4, 39, 65]. Regarding vulnerability locality, security experts and researchers tend to focus on hunting vulnerabilities in unsafe code of Rust packages [4, 19], leading to the potential increase in the ratio of unsafe code in vulnerable code of the Rust ecosystem.

7 RELATED WORK

7.1 Software Ecosystems

Prior research presents a steady stream of empirical studies on various software ecosystems, including JavaScript (npm) ecosystem [11, 26, 30, 64, 71], Python (PyPI) ecosystem [1, 5, 8, 23], and

[10] https://github.com/ZXXYy/rust_ecosystem

Java ecosystem [7, 38, 61]. In terms of the npm ecosystem, researchers investigate the package usage [11, 64], dependencies of packages [12, 26, 64, 71], and security risks in the ecosystem [30, 71]. The studies find that the npm ecosystem is steadily growing, with ongoing and accelerating growth in the number of packages and increasing dependencies between packages [64]; individual vulnerable packages could impact a large portion of the entire npm ecosystem [71]. As for the Python ecosystem, researchers characterize the developers of the ecosystem [23], growth in packages [8], dependencies of packages [1], and security risks in the ecosystem [1, 5]. The findings indicate that the Python ecosystem grows exponentially [8, 23]; the number of disclosed vulnerabilities in Python packages increases over time, and the vulnerabilities have been disclosed over 3 years after the relevant code was introduced in code repositories [1]. Studies on the Java ecosystem also explore the dependencies of packages [7], and the security risks in the Java ecosystem [38, 61].

A few studies investigate some aspects of the Rust ecosystem, by comparing it with other software ecosystems. Kikas et al.'s work [26] compared the structure and evolution of dependency networks across the JavaScript, Ruby, and Rust ecosystems. The reported results show that the analyzed ecosystems are alive and growing, with JavaScript having the fastest growth; software ecosystems are not as vulnerable to the removal of individual packages as they used to be. Serebrenik et al. [47] conducted a meta-analysis of the open challenges in software ecosystem research. As a result, they identified six open challenges, including quality and design, governance, dynamics and evolution, data analytics, domain-specific ecosystem solutions, and analysis of ecosystems.

Different from the studies above, we focus our research on the Rust ecosystem, by applying a systematic approach to investigating the security risks in the Rust ecosystem. Given dependencies of Rust packages have been investigated in a previous study [26], we do not include such analysis in our study.

7.2 Empirical Studies on Rust

In the past, researchers have conducted empirical studies on various aspects of Rust libraries and projects in practice, including the usage of Rust language features [2, 39] and bug characteristics [19, 65]. Some recent studies investigate the usage of unsafe Rust code in practice [2, 19]. Astrauskas et al. [2] analyze a large corpus of Rust projects to assess the validity of the Rust hypothesis, and classify the purposes of unsafe Rust code. Evans et al. [19] study how software developers use unsafe Rust code. Different from the studies on the usage of unsafe Rust code, our study investigates the relations between unsafe Rust code and disclosed vulnerabilities in the Rust ecosystem. A recent study [46] investigated the social risks of the Rust ecosystem from the perspective of developers and suggested ways for deploying limited developer resources to improve overall ecosystem health. Our work focuses more on the security risks in the Rust ecosystem from a technical perspective.

Other recent studies explore memory safety and concurrency issues in real-world Rust programs [39, 65]. Qin et al. [39] manually inspect 850 unsafe code usages, 70 memory safety bugs, and 100 thread safety bugs located in five open-source Rust-based systems and applications, and five widely-used Rust libraries. Xu et al. [65] focus their study on 186 Rust bug reports related to memory safety by December 31, 2020. They find that Rust can keep its memory safety promise, given that developers are unable to write memory-safety bugs without using unsafe code.

Different from Xu et al.'s work [65], our study does not focus on memory safety issues. Instead, our study involves a variety of types of vulnerabilities and makes an in-depth analysis across different types of vulnerabilities. In addition, we investigate the relation between vulnerabilities and code regions with `unsafe` keywords in vulnerable Rust packages. Different from Qin et al.'s work [39], our work investigates whether and how unsafe Rust code involves in vulnerabilities of the Rust ecosystem. Moreover, our dataset includes real-world Rust vulnerabilities of a variety of types, apart from the memory safety and concurrency issues as investigated in the prior work [39].

7.3 Securing Rust

Some research efforts have been invested in securing Rust software via formal verification [3, 6, 13, 25, 44, 62]. Patina [44] is a formalization of the Rust type system. Patina captures the key Rust features relevant to memory safety and specifies how the features guarantee memory safety. RustBelt [13, 25] defines rules to model Rust programs, and further uses these rules to prove the safety of Rust APIs. RustBelt also formally proves the memory safety of a realistic subset of Rust, including several standard Rust libraries with the existence of unsafe Rust. For a new Rust library that uses unsafe Rust code, RustBelt can tell the verification conditions that the library should meet to be considered as a safe extension to the Rust language. Baranowski et al. [6] extend the SMACK verifier [41], a software verification toolchain, to enable its usage on Rust programs. In addition, Astrauskas et al. [3] propose a verification technique that utilizes the type system of Rust to simplify the specification and verification of Rust programs. The technique can assist developers to verify their programs with formal methods.

Researchers also propose numerous techniques to detect bugs in Rust software programs [18, 29, 58]. Dewey et al. [18] propose a fuzzing testing approach to detect type-checker bugs of Rust programs. Toman et al. [58] introduce CRUST, a tool that combines exhaustive test generation and bounded model checking to verify memory safety in unsafe Rust code. The experimental results indicate that CRUST is effective at finding memory errors in the Rust standard libraries. Lindner et al. [29] propose a verification process for Rust programs via symbolic execution to detect unsafe and panic issues. Our work investigates the fix patterns of different types of vulnerabilities in the Rust ecosystem to shed lights on securing Rust software programs.

8 CONCLUSION AND FUTURE WORK

In this paper, we conducted a large-scale empirical study on the security risks of the Rust ecosystem, one of the emerging and growing software ecosystems aimed at the development of systems software. Specifically, we characterized the disclosed vulnerabilities, vulnerable packages affected by the vulnerabilities, and corresponding vulnerability fixes in the Rust ecosystem. We find that the vulnerabilities of different types differ widely in total numbers, disclosure and fixing duration, growth rates, and distributions across package categories in the Rust ecosystem. Among the 17 vulnerability types we identified, the memory safety and concurrency issues account for the majority of the disclosed vulnerabilities and demonstrate the fastest growth rates over time. One-third of the vulnerabilities have no fixes released by their public disclosure, leaving a window of opportunity for potential attacker exploitation. In the vulnerable packages, vulnerable code has statistically significantly higher ratios of unsafe functions and unsafe blocks compared to complete code, implying the potential higher security risks in unsafe functions and unsafe blocks. In addition, we identified three fix patterns for Rust vulnerabilities of different localities in Rust code.

Future work could consider developing an automatic tool to continuously collect and analyze the packages and vulnerabilities in the Rust ecosystem, and further leverage the real-time analysis results to gain awareness of the security risks in the Rust ecosystem in a timely way.

REFERENCES

[1] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2021. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 446–457.

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

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

[4] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3477132.3483570

[5] Aadesh Bagmar, Josiah Wedgwood, Dave Levin, and Jim Purtilo. 2021. I Know What You Imported Last Summer: A study of security threats in thePython ecosystem. *arXiv preprint arXiv:2102.06301* (2021).

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

[7] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015), 1275–1317.

[8] Ethan Bommarito and Michael J. Bommarito II. 2019. An Empirical Analysis of the Python Package Index (PyPI). *CoRR* abs/1907.11073 (2019). arXiv:1907.11073 http://arxiv.org/abs/1907.11073

[9] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2021. Neural transfer learning for repairing security vulnerabilities in c code. *arXiv preprint arXiv:2104.08308* (2021).

[10] Yong Wen Chua. 2017. Appreciating Rust's Memory Safety Guarantees. *Government Digital Services, Singapore. July* 14 (2017).

[11] Filipe R Cogo, Gustavo A Oliva, Cor-Paul Bezemer, and Ahmed E Hassan. 2021. An empirical study of same-day releases of popular packages in the npm ecosystem. *Empirical Software Engineering* 26, 5 (2021), 1–42.

[12] Filipe Roseiro Cogo, Gustavo A. Oliva, and Ahmed E. Hassan. 2021. An Empirical Study of Dependency Downgrades in the npm Ecosystem. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2457–2470. https://doi.org/10.1109/TSE.2019.2952130

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

[14] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the Topology of Package Dependency Networks: A Comparison of Three Programming Language Ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops* (Copenhagen, Denmark) *(ECSAW '16)*. Association for Computing Machinery, New York, NY, USA, Article 21, 4 pages. https://doi.org/10.1145/2993412.3003382

[15] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2–12. https://doi.org/10.1109/SANER.2017.7884604

[16] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. 2016. When GitHub Meets CRAN: An Analysis of Inter-Repository Package Dependency Problems. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 493–504. https://doi.org/10.1109/SANER.2016.12

[17] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

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

[19] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is Rust Used Safely by Software Developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 246–257. https://doi.org/10.1145/3377811.3380413

[20] Stefan Frei. 2011. End-Point Security Failures, Insight gained from Secunia PSI scans. In *Predict Workshop, February*.

[21] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. In *NDSS*.

[22] Daniel M. German, Bram Adams, and Ahmed E. Hassan. 2013. The Evolution of the R Software Ecosystem. In *2013 17th European Conference on Software Maintenance and Reengineering*. 243–252. https://doi.org/10.1109/CSMR.2013.33

[23] Rick Hoving, Gabriel Slot, and Slinger Jansen. 2013. Python: Characteristics identification of a free open source software ecosystem. In *2013 7th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*. IEEE, 13–18.

[24] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *2016 IEEE Symposium on Security and Privacy (SP)*. 618–635. https://doi.org/10.1109/SP.2016.43

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

[26] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 102–112.

[27] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2201–2215. https://doi.org/10.1145/3133956.3134072

[28] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2021. MirChecker: detecting bugs in Rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2183–2196.

[29] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No panic! Verification of Rust programs by symbolic execution. In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, 108–114.

[30] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. *arXiv preprint arXiv:2201.03981* (2022).

[31] Nikolai Mansourov and Djenana Campara. 2010. *System Assurance: Beyond Detecting Vulnerabilities* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2013. The design of bug fixes. In *2013 35th International Conference on Software Engineering (ICSE)*. 332–341. https://doi.org/10.1109/ICSE.2013.6606579

[33] Stack Overflow. 2016. Stack Overflow Developer Survey 2016. (2016). https://insights.stackoverflow.com/survey/2016#technologymost-loved-dreaded-and-wanted

[34] Stack Overflow. 2017. Stack Overflow Developer Survey 2017. (2017). https://insights.stackoverflow.com/survey/2017#technologymost-loved-dreaded-and-wanted

[35] Stack Overflow. 2018. Stack Overflow Developer Survey 2018. (2018). https://insights.stackoverflow.com/survey/2018#technologymost-loved-dreaded-and-wanted

[36] Alex Ozdemir. 2022. Unsafe in Rust: Syntactic Patterns. (2022). https://cs.stanford.edu/~aozdemir/blog/unsafe-rust-syntax/

[37] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. 2012. An empirical study of supplementary bug fixes. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. 40–49. https://doi.org/10.1109/MSR.2012.6224298

[38] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering* (2020).

[39] 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 Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 763–779. https://doi.org/10.1145/3385412.3386036

[40] Quantum. 2019. Quantum. (2019). https://wiki.mozilla.org/Quantum

[41] Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling source language details from verifier implementations. In *International Conference on Computer Aided Verification*. Springer, 106–113.

[42] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 428–439.

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

[44] Eric Reed. 2015. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* (2015), 264.

[45] Maven Repository. 2022. Maven Repository: Open Source. (2022). https://mvnrepository.com/open-source

[46] Willam Schueller and Johannes Wachs. 2022. Modeling Interconnected Social and Technical Risks in Open Source Software Ecosystems. *arXiv preprint arXiv:2205.04268* (2022).

[47] Alexander Serebrenik and Tom Mens. 2015. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. 1–6.

[48] Servo. 2019. The Servo Browser Engine. (2019). https://servo.org/

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

[50] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[51] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1275–1295. https://doi.org/10.1109/SP.2019.00010

[52] Mauricio Soto, Ferdian Thung, Chu-Pan Wong, Claire Le Goues, and David Lo. 2016. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 512–515.

[53] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python Framework for Mining Software Repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911. https://doi.org/10.1145/3236024.3264598

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

[55] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*. 48–62. https://doi.org/10.1109/SP.2013.13

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

[57] Tock. 2019. Tock Embedded Operating System. (2019). .https://www.tockos.org/

[58] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: a bounded verifier for rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 75–80.

[59] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 832–837.

[60] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.

[61] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.

[62] Aaron Weiss, Daniel Patterson, and Amal Ahmed. 2018. Rust distilled: An expressive tower of languages. *arXiv preprint arXiv:1806.02693* (2018).

[63] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[64] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A Look at the Dynamics of the JavaScript Package Ecosystem. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. 351–361.

[65] 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. 31, 1, Article 3 (sep 2021), 25 pages. https://doi.org/10.1145/3466642

[66] Ruru Yue, Na Meng, and Qianxiang Wang. 2017. A Characterization Study of Repeated Bug Fixes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 422–432. https://doi.org/10.1109/ICSME.2017.16

[67] Shahed Zaman, Bram Adams, and Ahmed E Hassan. 2011. Security versus performance bugs: a case study on firefox. In *Proceedings of the 8th working conference on mining software repositories*. 93–102.

[68] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and RubyGems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 1–45.

[69] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 913–923.

[70] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.

[71] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.