RustXec: A Vulnerability Reproduction Dataset for Assessing Security Risks in Open-Source Rust Applications

Zhengjie Ji
zhengjie@vt.edu
Virginia Tech
United States

Xin Wang
xinw@vt.edu
Virginia Tech
United States

Wang Lingxiang
lingxiang.wang.2016@gmail.com
Unaffiliated
United States

Geng Li
lig25@wfu.edu
Wake Forest University
United States

Fan Yang
yangfan@wfu.edu
Wake Forest University
United States

Ying Zhang*
ying.zhang@wfu.edu
Wake Forest University
United States

Abstract

Despite Rust's memory safety guarantees, developers can still introduce security vulnerabilities due to limited security awareness and training. Assessing the security risks of such vulnerabilities is challenging, especially when the resulting failures are not directly observable in the application's runtime behavior. However, the Rust ecosystem currently lacks reproducible vulnerability datasets, and many vulnerability advisories do not provide proof-of-vulnerability (PoV) examples to demonstrate the issue. As a result, reproducing vulnerabilities from advisory information alone is technically difficult and time-consuming, which limits developers' ability to recognize and understand security risks in practice.

In this paper, we built RustXec, a comprehensive reproducible dataset for Rust vulnerabilities. RustXec includes 102 vulnerabilities across 89 open-source Rust projects. It covers eight vulnerability categories collected from the RustSec security advisory database. For each vulnerability advisory, RustXec includes a verified PoV that demonstrates the security flaw as described in the vulnerability advisories, along with a containerized execution environment to facilitate the reproduction. This dataset enables developers and researchers to reliably reproduce vulnerabilities and understand the security risk in Rust applications.

ACM Reference Format:
Zhengjie Ji, Xin Wang, Wang Lingxiang, Geng Li, Fan Yang, and Ying Zhang. 2026. RustXec: A Vulnerability Reproduction Dataset for Assessing Security Risks in Open-Source Rust Applications. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil*. International Conference on Mining Software Repositories, Rio de Janeiro, Brazil, 5 pages. https://doi.org/10.1145/3793302.3793307

1 Introduction

Rust is a programming language designed to ensure memory safety at compile time through its ownership and borrowing system [5–7, 17]. Prior work [11, 26] has shown that developers can still introduce security vulnerabilities into Rust systems when using `unsafe` blocks, mismanaging memory, or making logic flaws during the implementation. These vulnerabilities are often *invisible* to developers and may remain undetected unless particular runtime conditions trigger observable failures. As a result, developers frequently struggle to understand how a reported vulnerability can be exploited and to assess its security implications from vulnerability reports alone [15]. When overlooked, such vulnerabilities can result in data corruption, degrade system reliability, and even cause severe financial losses [12, 13].

Despite the prevalence and potential impact of these vulnerabilities, there is still a lack of vulnerability datasets to help developers assess security risks in Rust applications. Existing Rust vulnerability datasets collected in the prior work [8, 10, 14, 16, 19, 20, 23, 24, 27] either include non-reproducible vulnerabilities [24, 27] or only cover a single vulnerability category [8, 10, 14, 16]. For example, Yuga [16] uses a comparatively small dataset of nine crates from RustSec, and focuses exclusively on memory-related vulnerabilities. Additionally, the vulnerability datasets collected by Xu et al. [24] and Zheng et al. [27] provide general vulnerability descriptions from CVE [2] or RustSec advisories [21]. They do not include executable test cases or scripts, so-called proof-of-vulnerabilities (PoVs), which demonstrate how vulnerabilities can be triggered. The absence of a reproducible, multi-category vulnerability dataset for runtime observation makes it challenging to assess security risks and hinders the adoption of secure coding practices in Rust applications.

Reproducing vulnerabilities in Rust applications is challenging due to technical complexity and time demands. First, without PoVs, developers or security practitioners are required to examine the project source code, identify triggering conditions, and derive concrete inputs that satisfy the necessary data and execution path constraints [25]. Second, observing runtime issues for a given vulnerability requires setting up precise runtime environments and configuring project dependencies for each Rust project. Additionally, developers should verify that the program execution behavior observed during execution corresponds to the target vulnerability, which is both labor-intensive and time-consuming.

In this paper, we build RustXec, a vulnerability reproduction dataset for the Rust open-source ecosystem. RustXec includes 102 reproducible Rust vulnerabilities across eight categories, which correspond to 89 open-source Rust projects. To build the dataset, we began by crawling entries from the RustSec security advisory

*Corresponding author

MSR '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793307

database [21], a community-maintained platform that tracks vulnerabilities in crates.io packages. We chose RustSec because it focuses on open-source Rust crates and serves as the source for GitHub Advisory's Rust vulnerabilities [3], whereas NVD [4] includes closed-source applications and kernel code outside our scope. Given the significant manual effort required to reproduce vulnerabilities across different Rust versions, we limited our scope to 2021-2025, which yielded 515 initial vulnerability entries. From these entries, we first filtered out 216 entries that are vulnerability-irrelevant. Second, we extracted PoV candidates and their affected projects from the advisory references, which include vulnerability descriptions, fix commits, and discussion in pull requests. We excluded projects that were not compilable or lacked any PoV candidates, reducing our dataset to 107 vulnerability entries. Third, we executed each PoV candidate on the affected project and manually verified the observed behavior against the vulnerability description in the advisory; we confirmed 102 vulnerabilities with verified PoVs, which form the final dataset. In order to facilitate the dataset reuse and reduce environment setup time, we constructed a containerized environment with a verified PoV for each vulnerability.

In summary, our paper makes the following contributions:

- We build the first real-world dataset that includes 102 reproducible vulnerabilities with verified PoVs, which covers eight different vulnerability categories in the 89 Rust open-source projects.
- We provide a ready-to-use containerized environment that eliminates the complexity of environment configuration, allowing developers to easily reproduce vulnerabilities without manual configuration.

RustXec is available at: https://github.com/ying-selab/RustXec.

2 Methodology

This section introduces the two-stage workflow to build RustXec (as shown in Figure 1). In the data preparation stage (Section 2.1), we collect and filter vulnerabilities from RustSec. We then identify PoV candidates and verify that affected projects are compilable and runnable. In the vulnerability reproduction stage (Section 2.2), we apply PoV candidates to affected projects and execute them. Next, we confirm whether each vulnerability is reproduced by matching the observed program execution behavior with its vulnerability description. Finally, we package each reproduced vulnerability as an artifact, including the project and the PoV that triggers the vulnerability as a Docker image, together with the relevant vulnerability metadata (Section 2.3).

2.1 Data Preparation

Our data preparation process aims to systematically collect: (1) open-source Rust projects containing known security vulnerabilities. To ensure these projects can be used in the vulnerability reproduction stage, each project must be compilable in a controlled environment. (2) PoV candidates that can *potentially* demonstrate the vulnerabilities in these Rust projects.

Vulnerability entries collection and filtering. We collect vulnerabilities from the RustSec security advisory database [21], as it is the most comprehensive community-driven vulnerability database specifically for the Rust ecosystem. RustSec tracks security issues

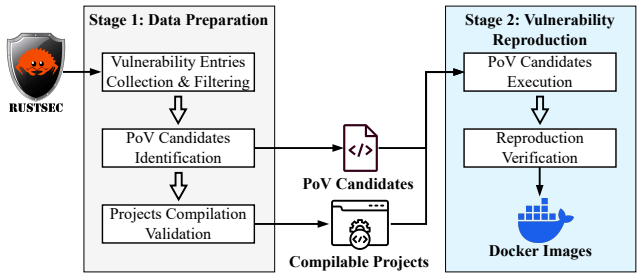

Figure 1: Dataset construction workflow for RustXec.

in open-source Rust crates on `crates.io`, providing structured information (e.g., affected functions) that supports vulnerability reproduction and analysis. We choose RustSec over databases like NVD [4] because RustSec focuses exclusively on open-source Rust crates, while NVD also includes closed-source software vulnerabilities that cannot be reproduced. We collect 515 vulnerabilities reported between January 2021 and April 2025. RustXec only includes vulnerabilities after 2021, because the recent Rust compiler introduced significant improvements that enhanced compile-time checks, making many pre-2021 vulnerability patterns detectable. For each advisory, we automatically extract vulnerability information including vulnerability category, patched versions, and affected functions mentioned in the vulnerability description. To ensure each case in RustXec is vulnerability related, we exclude advisories marked as *unmaintained* (117 cases) or *unsound* (89 cases) and remove duplicate cases that refer to the same underlying issue (10 cases). After the filtering process, there are 299 cases remaining.

PoV candidates identification. For each advisory, we collect potential PoVs that may demonstrate the existence of vulnerability. A PoV refers to inputs, test cases, or scripts that can potentially trigger the vulnerability in the affected project [9]. Our insight is that PoVs can exist within the following sources: (1) vulnerability descriptions in RustSec advisories; (2) vulnerability fix commits, developers may introduce unit tests to validate the patch and confirm vulnerability removal; and (3) discussions in GitHub issues and pull requests for fixes. Therefore, we extracted all potential PoVs from these sources. In total, we identified 154 PoV candidates corresponding to 118 RustSec advisories, which included 16 from RustSec advisory descriptions, 84 from fix commits, and 54 from GitHub discussions (issues and vulnerability fix pull requests).

Projects with known vulnerability. To collect vulnerability projects for reproduction, we applied two criteria: 1) the affected project has PoV candidates, and 2) it must be compilable under the vulnerable version and configuration described in the vulnerability entries. We first excluded 181 projects without PoV candidates. We then cloned the remaining 118 projects from GitHub or GitLab, checked out commits prior to the fix, and set the Rust version accordingly. We applied `cargo build` for each affected project and resolved dependency issues (such as version conflicts). Since our reproduction system environment is Ubuntu 24.04, we further excluded 11 projects that require specific hardware (e.g., ARM architecture) or OS-specific features (e.g., Windows-only APIs). We also followed the README file to execute the project and confirmed it was runnable. For library projects, we only applied `cargo build`

without running, because they do not have a main function. As some projects contained multiple vulnerabilities at different versions, after merging the duplication, we got 94 unique projects corresponding to 107 vulnerability entries.

2.2 Vulnerability Reproduction

We verify vulnerability reproduction by executing PoV candidates on their corresponding affected projects. We then manually inspect the program's runtime behavior against the vulnerability description to identify the PoV that successfully triggers the vulnerability. We refer to these confirmed triggering inputs as *verified PoVs*.

PoV candidates execution. We configure the execution environment according to different types of PoVs. (1) For unit test PoVs (59 cases), which are specific test functions or code snippets, we execute them directly using `cargo test` as part of the project's unit testing framework. (2) For script-based PoVs (48 cases) that require interaction with a client project or external environment, we manually construct the environment, configure the dependencies, and execute them as independent programs. For example, RUSTSEC-2024-0376 describes a remotely exploitable denial-of-service vulnerability in the Tonic gRPC framework. We set up both server and client components in the PoV: the PoV script first starts the Tonic server with TLS enabled, then launches a client configured with an incorrect domain name (e.g., "wrong.com" instead of "example.com").

To provide sufficient information for determining whether a PoV successfully reproduces the vulnerability, we capture the program runtime behavior during the execution. Specifically, we use the `script` and `top` command to record program output, abnormal termination signals, and capture resource consumption (CPU and memory usage). In addition, we re-run the executable with `rust-gdb` independently to extract stack traces.

Reproduction verification. To obtain verified PoVs, we first evaluate each candidate by checking whether the program output and abnormal termination signals match the advisory description. We further use the recorded stack traces to confirm that the vulnerable code path is executed and the vulnerability is successfully triggered. Specifically, for different vulnerability category, we need extra specific information to validate reproduction. For example, regarding memory-corruption and memory-exposure vulnerabilities, we examine AddressSanitizer [22] output alongside program output to confirm detection of memory safety violations (e.g., buffer overflows, use-after-free); we consider the vulnerability reproduced if AddressSanitizer reports errors or if the program crashes with segmentation faults or panics. For denial-of-service vulnerabilities, reproduction is confirmed if the PoV candidate fails to complete within the timeout or exhibits abnormal resource consumption, such as excessive CPU usage or memory exhaustion.

Through this validation process, we successfully reproduced 102 out of 107 vulnerabilities with verified PoVs. The remaining five cases failed to be reproduced as their advisories lack runtime configurations or environmental prerequisites essential for reproduction.

2.3 Vulnerability Artifact Construction

To facilitate reproducibility, each vulnerability entry provides a containerized execution environment and the associated vulnerability metadata. The environment includes a `Dockerfile`, a pre-built `Docker` image, and the following data: (1) the source code of the affected project with the verified PoV; (2) all vendored Rust dependencies; (3) a `Makefile` that automates the build and reproduction process; and (4) example execution outputs demonstrating that the vulnerability has been triggered. Additionally, we provide metadata for all cases, such as categories, severity, patched versions, and affected functions. In summary, the standardized container images and comprehensive metadata collectively guarantee the accurate and efficient reproduction of the dataset's vulnerabilities.

3 Statistics

This section presents key statistics of RustXec to demonstrate the dataset's scope and characteristics. We analyze the distribution of vulnerabilities across different categories and years, examine the time requirements for vulnerability reproduction, and evaluate PoV execution performance.

Category distribution. RustXec contains 102 reproducible vulnerabilities from 89 distinct Rust projects, spanning eight vulnerability categories. Figure 2 illustrates how these vulnerability categories are distributed from 2021 to 2025. RustXec includes a relatively higher number of vulnerabilities in 2021 and 2024, with fewer cases in 2022 and 2023. The highest counts are memory-corruption (20 cases) in 2021 and denial-of-service (13 cases) in 2024. The remaining categories have relatively even distribution across years.

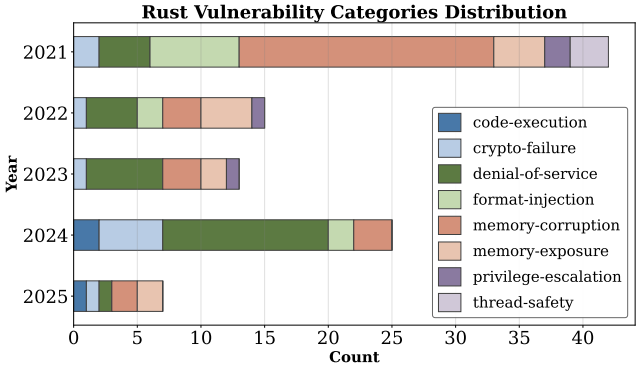

Figure 2: Distribution of vulnerabilities in RustXec by vulnerability category from January 2021 to April 2025.

RustXec contains 62 different Common Weakness Enumeration (CWE) [1] IDs (as shown in Table 1). We extract CWE IDs from CVE [2] and GHSA [3] entries and map them to RustSec categories. For cases without available CWE information, we omit this field. The most prevalent categories are memory-corruption (31 cases) and denial-of-service (28 cases), reflecting persistent challenges in memory safety and resource management. The most frequent CWEs are CWE-787 (out-of-bounds write; nine cases), CWE-415 (double free; eight cases), and CWE-400 (uncontrolled resource consumption; seven cases). This breadth supports comprehensive analyses of vulnerability characteristics across the Rust ecosystem.

Vulnerability reproduction effort. We spent 324 person-hours verifying and reproducing 102 vulnerabilities in RustXec. The effort required for reproduction varied substantially across vulnerability categories. Specifically, we found that vulnerability in

Table 1: Distribution of vulnerabilities in RustXec by RustSec category and corresponding CWE IDs. CWE IDs are obtained from the references in each RustSec advisory.

RustSec Category	Corresponding CWE IDs	# Cases
code-execution	23, 77, 88	3
crypto-failure	287, 288, 327, 328, 331, 347, 440, 497, 682, 863	10
denial-of-service	20, 130, 228, 232, 240, 248, 392, 400, 401, 476, 617, 670, 674, 754, 755, 770, 824, 835, 1284, 1333	28
format-injection	79, 113, 147, 436, 444, 601	11
memory-corruption	119, 120, 190, 191, 366, 415, 416, 787, 843, 908	31
memory-exposure	125, 126, 212, 226, 459, 590	12
privilege-escalation	22, 59, 269, 284, 668, 706	4
thread-safety	362	3
Total	**62 (# Distinct CWE IDs)**	**102**

privilege-escalation and code-execution required more time because they often depend on specific system configurations and permission settings. For example, RUSTSEC-2023-0066 is a privilege-escalation vulnerability in the pleaser crate that abuses the TIOCSTI and TIOCLINUX ioctl request codes to inject input into a terminal or trigger console-level commands. To reproduce it, we need to enable legacy `ioctl` support via kernel parameters changes, configure `setuid` root permissions, as well as compile a C exploit program to trigger the privilege escalation. Because each vulnerability can require substantial manual configuration, we provide reproduction instructions and a prebuilt Docker image for every vulnerability. Users can run a single command to reproduce and observe the runtime consequences.

PoV execution time. Executing the verified PoV to trigger the vulnerability takes 9.4 seconds on average (as shown in Table 2). Categories that cause immediate crashes or detectable runtime errors execute quickly, typically within a few seconds. For example, code-execution vulnerabilities execute in 0.4 seconds on average because the PoV triggers immediate, observable effects when executed in the shell (e.g., RUSTSEC-2024-0350). In contrast, categories requiring timeout waiting, resource monitoring, or intensive computations take longer. For example, crypto-failure vulnerabilities take 19.6 seconds on average as they involve computationally expensive cryptographic operations for exploitation (e.g., RUSTSEC-2022-0093).

Table 2: Average PoV execution time by vulnerability category in RustXec.

RustSec Category	Time (s)	RustSec Category	Time (s)
code-execution	0.4	memory-corruption	3.5
crypto-failure	19.6	memory-exposure	10.8
denial-of-service	12.0	privilege-escalation	6.5
format-injection	13.8	thread-safety	5.3
Overall			**9.4**

4 Threats to Validity

Our manual verification of PoV execution may introduce human bias. To mitigate this risk, we defined explicit reproduction criteria and had multiple authors independently validate the results. Additionally, we excluded vulnerabilities without available PoV candidates, which may limit the generalizability of our findings. Future work could broaden coverage by constructing PoVs based on information from vulnerability advisories. Moreover, our tool currently uses Ubuntu 24.04 as Docker image, which will reach end-of-life in 2029. To address this, we plan to update the Docker image to newer Ubuntu long time support versions as they become available, ensuring long-term tool availability.

5 Related Work

Several empirical studies have collected Rust vulnerability datasets to analyze vulnerability characteristics (e.g., memory bugs [24], concurrency bugs [19, 20]) and safety requirements for unsafe Rust programming [11]. While these studies provide valuable insights into Rust vulnerability patterns, many of them could not ensure the reproducibility of vulnerabilities. In contrast, our work guarantees reproducibility by providing a verified PoV along with a containerized environment, enabling developers to observe concrete program execution behavior and understand the real-world security impact.

Prior works have built Rust vulnerability datasets [10, 14, 16] alongside detection tools to support tool evaluation. TypePulse [10] identified 26 type confusion bugs from RustSec, which are undefined behaviors caused by unsafe type conversion operations in Rust. ERASan [14] introduced a dataset of 28 memory bugs to evaluate their customized address sanitizer tailored for the Rust environment. While these datasets focus on specific vulnerability types for evaluating specialized tools, RustXec provides a dataset covering eight vulnerability categories, enabling cross-category security analysis and supporting the evaluation of general-purpose Rust security tools.

There are several reproducible vulnerability datasets [9, 18] constructed for other programming languages. Bui et al. [9] constructed Vul4J, a dataset of 79 reproducible vulnerabilities for Java collected from Project KB, to support program repair research. Pinconschi et al. [18] proposed CB-REPAIR, a dataset containing 55 vulnerabilities in Linux-based C/C++ programs for evaluating automatic program repair techniques. Similar to these efforts, our work provides the first comprehensive reproducible vulnerability dataset for Rust. This fills a critical gap in Rust security research and advances secure coding practices in the growing Rust community.

6 Conclusion

This paper introduces RustXec, the first comprehensive reproducible Rust vulnerability dataset for assessing security risks in Rust applications. RustXec contains 102 vulnerabilities spanning eight vulnerability categories, each with a verified PoV and detailed vulnerability information for reproduction and analysis. To facilitate easy reuse, RustXec provides containerized environments that eliminate manual environment setup effort. In the future, we will leverage RustXec to provide richer evaluation signals for systematically analyzing existing Rust vulnerability detection tools and guide the design of new detection tools.

References

[1] 2025. Common Weakness Enumeration. https://cwe.mitre.org/. [Online; accessed 2025-09-06].

[2] 2025. CVE: Common Vulnerabilities and Exposures. https://www.cve.org/. [Online; accessed 2025-10-15].

[3] 2025. GitHub Advisory Database. https://github.com/advisories. [Online; accessed 2025-10-06].

[4] 2025. National Vulnerability Database. https://nvd.nist.gov/. [Online; accessed 2025-10-06].

[5] 2025. Rust for Linux. https://rust-for-linux.com/. [Online; accessed 2025-09-20].

[6] 2025. The Rust Programming Language. https://doc.rust-lang.org/stable/book/. [Online; accessed 2025-09-20].

[7] 2025. The Rustonomicon. https://doc.rust-lang.org/nomicon/. [Online; accessed 2025-09-20].

[8] 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*. 84–99.

[9] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 464–468. doi:10.1145/3524842.3528482

[10] Hung-Mao Chen, Xu He, Shu Wang, Xiaokuan Zhang, and Kun Sun. 2025. TYPEPULSE: Detecting Type Confusion Bugs in Rust Programs. *arXiv preprint arXiv:2502.03271* (2025).

[11] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[12] Extropy.IO. 2022. Solana's Wormhole Hack Post-Mortem Analysis. https://extropy-io.medium.com/solanas-wormhole-hack-post-mortem-analysis-3b68b9e88e13.

[13] Matthew Prince. 2025. Cloudflare outage on November 18, 2025. https://blog.cloudflare.com/18-november-2025-outage/.

[14] Jiun Min, Dongyeon Yu, Seongyun Jeong, Dokyung Song, and Yuseok Jeon. 2024. ERASan: Efficient Rust Address Sanitizer. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4053–4068. doi:10.1109/SP54263.2024.00258

[15] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*. 919–936.

[16] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. 2024. Yuga: Automatically Detecting Lifetime Annotation Bugs in the Rust Language. *IEEE Trans. Softw. Eng.* 50, 10 (Oct. 2024), 2602–2613. doi:10.1109/TSE.2024.3447671

[17] Yuke Peng, Hongliang Tian, Zhang Junyang, Ruihan Li, Chengjun Chen, Jianfeng Jiang, Jinyi Xian, Xiaolin Wang, Chenren Xu, Diyu Zhou, Yingwei Luo, Shoumeng Yan, and Yinqian Zhang. 2025. Asterinas: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB. arXiv:2506.03876 [cs.OS] https://arxiv.org/abs/2506.03876

[18] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A Comparative Study of Automatic Program Repair Techniques for Security Vulnerabilities. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 196–207. doi:10.1109/ISSRE52982.2021.00031

[19] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and detecting real-world safety issues in rust. *IEEE Transactions on Software Engineering* (2024).

[20] 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*. 763–779.

[21] RustSec Project. 2025. RustSec Advisory Database. https://rustsec.org/. [Online; accessed 2025-07-03].

[22] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[23] Diane B Stephens, Kawkab Aldoshan, and Mustakimur Rahman Khandaker. 2024. Understanding the Challenges in Detecting Vulnerabilities of Rust Applications. In *2024 IEEE Secure Development Conference (SecDev)*. IEEE, 54–63.

[24] 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 (Sept. 2021), 25 pages. doi:10.1145/3466642

[25] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. 2022. Automatic detection of Java cryptographic API misuses: Are we there yet? *IEEE Transactions on Software Engineering* 49, 1 (2022), 288–303.

[26] Yuchen Zhang, Ashish Kundu, Georgios Portokalidis, and Jun Xu. 2023. On the dual nature of necessity in use of Rust unsafe code. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2032–2037.

[27] Xiaoye Zheng, Zhiyuan Wan, Yun Zhang, Rui Chang, and David Lo. 2023. A Closer Look at the Security Risks in the Rust Ecosystem. *ACM Trans. Softw. Eng. Methodol.* 33, 2, Article 34 (Dec. 2023), 30 pages. doi:10.1145/3624738