Can Secure Software be Developed in Rust? On Vulnerabilities and Secure Coding Guidelines

Tiago Espinha Gasiba
T CST SEL-DE
Siemens AG
Munich, Germany
tiago.gasiba@siemens.com

Sathwik Amburi
T CST SEL-DE, Siemens AG
Technical University of Munich
Munich, Germany
sathwik.amburi@{siemens.com, tum.de}

Andrei-Cristian Iosif
T CST SEL-DE
Siemens AG
Munich, Germany
andrei-cristian.iosif@siemens.com

***Abstract*—Since the Rust programming language was accepted into the Linux Kernel, it has gained significant attention from the software developer community and the industry. Rust has been developed to address many traditional software problems, such as memory safety and concurrency. Consequently, software written in Rust is expected to have fewer vulnerabilities and be more secure. However, a systematic analysis of the security of software developed in Rust is still missing. The present work aims to close this gap by analyzing how Rust deals with typical software vulnerabilities. We compare Rust to C, C++, and Java, three widely used programming languages in the industry, regarding potential software vulnerabilities. We also highlight ten common security pitfalls in Rust programming that we think software developers and stakeholders alike should be wary of. Our results are based on a literature review and interviews with industrial cybersecurity experts. We conclude that, while Rust improves the status quo compared to the other programming languages, writing vulnerable software in Rust is still possible. Our research contributes to academia by enhancing the existing knowledge of software vulnerabilities. Furthermore, industrial practitioners can benefit from this study when evaluating the use of different programming languages in their projects.**

Index Terms*—*Cybersecurity*; *Software development*; *Industry*; *Software*; *Vulnerabilities*; *Rust Programming Language.

I. INTRODUCTION

Rust, a systems programming language that originated in 2010, has significantly increased in popularity over the past decade. Rust distinguishes itself from other programming languages through several key features. Firstly, it ensures memory safety without needing a garbage collector, utilizing an ownership system with rules about borrowing and lifetimes. This feature helps prevent common bugs such as null pointer dereferencing and dangling pointers prevalent in languages like C and C++. Secondly, Rust's concurrency model is designed to make concurrent programming safer and more straightforward, with the ownership and type systems enforcing thread safety and preventing data races at compile time. Additionally, Rust provides performance comparable to C and C++ due to its low-level control over system resources and zero-cost abstractions. Unlike higher-level languages such as Java, which rely on a virtual machine, Rust compiles directly to machine code, offering predictable performance and minimal runtime overhead. Rust's expressive type system supports advanced features like algebraic data types, pattern matching, and traits (Rust's version of interfaces), enabling developers to write robust and maintainable code. Moreover, Rust's tooling and ecosystem, including the Cargo package manager, facilitate dependency management and project building, making the development process smoother and more efficient. The Rust community is also known for its welcoming nature and comprehensive documentation, providing extensive learning materials for developers.

This paper extends the authors' previous work presented at the CYBER 2023 [1], which covered the initial findings and methodologies. The current study includes some additional insights into the Rust programming language and common security pitfalls.

According to a market overview survey by Yalantis [2], which conducted more than 9,300 interviews, 89% of developers prefer Rust over other widespread programming languages like C and C++ due to its robust security properties. Despite its steep learning curve, industry professionals argue that the time invested in learning Rust yields added benefits and fosters better programming skills [3]. Stack Overflow notes that developers appreciate Rust's focus on system-level details, which helps prevent null and dangling pointers, and its memory safety without the need for a garbage collector [4]. These factors contribute to its growing adoption in the industry. This sentiment is echoed by the industry's push toward adopting the Rust programming language. Furthermore, according to Stack Overflow Developer Surveys, Rust has been the most loved and admired language since 2016. In the most recent Stack Overflow 2023 Developer Survey [5], Rust secured the position of the most admired language, with over 80% of the 87,510 responses favoring it.

Due to its focus on memory safety and concurrency, Rust has become the language of choice for many tools developed for Linux, FreeBSD, and other operating systems. Notably, Rust's adoption in Linux Kernel development [6], [7] underscores its growing significance in an industrial context, including space systems [8]. Rust meets the critical requirements for such applications in several key aspects. First, its design enables code to operate close to the kernel, facilitating tight interactions with both software and hardware. Second, it supports determinism, ensuring consistent timing of outputs. Third, Rust does not use a garbage collector, which is crucial

for manual memory management and reclaiming memory without automatic interference. These characteristics ensure reliable and predictable outcomes, vital for space systems [8] and other critical industrial systems. Despite these advantages, Rust's proven effectiveness in space systems remains to be fully demonstrated. Developing toolchains, workforce education, and case studies in field applications are vital to validating the utility of memory-safe languages in such demanding environments [8].

Major platforms, such as Google, have started including Rust in their systems, including Android [9], demonstrating its broad applicability. Additionally, forums like RustSec [10] offer real-time updates and insights into the current state of Rust security, reflecting its critical role in secure computing. Recognizing these advantages, governments, including the US, have begun recommending memory-safe languages like Rust [11], further emphasizing its importance in terrestrial and extraterrestrial computing environments.

Rust promotes itself as being safer than traditional languages, such as C and C++, which are widely used in an industrial context, by borrowing many aspects from functional languages like Haskell. However, in the realm of industry, particularly in critical infrastructures, safety is not synonymous with security. As the industry is obliged to follow secure development standards, such as IEC 62443 [12], [13], the notion of safety in Rust must be understood not only from a memory management perspective but also from a security standpoint [14]. Rust was developed to address memory-related vulnerabilities, which constitute only 19.5% of the most exploited vulnerabilities in 2023, according to The Cybersecurity & Infrastructure Security Agency (CISA) [15]. Exploits related to routing and path abuse tied for second place with memory vulnerabilities, followed by default secrets (4.9%), request smuggling (4.9%), and weak encryption (2.4%). The most prevalent exploit was insecure exposed functions (IEF), accounting for 48.8% of incidents. This has led to the saying, "Rust won't save us, but its ideas will," emphasizing the importance of adopting Rust's safety and security principles beyond its direct application [16].

Developing industrial products and services follows strict guidelines, especially for those products and services aimed at critical infrastructures. In these cases, cybersecurity incidents can severely negatively impact companies and society in general. Therefore, the security of industrial products must be tightly controlled. Consequently, Rust is considered a good candidate for industrial software development.

While Rust has been celebrated for its safety features [17], [18], less research has been conducted on its security aspects. This lack of research is primarily because this programming language is still relatively young compared to longstanding players in the industry, such as C, C++, and Java. Furthermore, developers and users often conflate safety with security, potentially leading to software vulnerabilities. Therefore, this paper aims to *understand to what extent vulnerable software can be written in Rust*. We approach this topic in two ways:

1) Evaluating the difficulty of writing vulnerable software based on industry-recognized security standards like the SysAdmin, Audit, Network, Security (SANS) Institute TOP 25 [19], the Open Web Application Security Project (OWASP) TOP 10 [20], and the 19 Deadly Sins [21], and
2) Identifying ten common pitfalls in Rust that we feel developers should be aware of.

This study's contributions are as follows: firstly, through the present work, the authors aim to raise awareness, as defined by Gasiba et al. [22], about Rust security and its pitfalls within the industry (for both industrial practitioners and academia); secondly, our work provides expert opinions from industry security experts on how to mitigate such issues when developing software with Rust; furthermore, our work contributes to academic research and the body of knowledge on Rust security by adding new insights and fostering a deeper understanding of Rust security; finally, our work serves as motivation for further studies in this area.

The rest of this paper is organized as follows: Section II discusses previous work that is either related to or served as inspiration for our study. Section III briefly discusses the methodology followed in this work to address the research questions. In Section IV, we provide a summary of our results, and in Section V, we conduct a critical discussion of these results. Finally, in Section VI, we conclude our work and outline future research.

II. RELATED WORK

A significant contribution to understanding Rust's security model comes from Sible et al. [23]. Their work offers a thorough analysis of Rust's security model, focusing on its memory and concurrency safety features. However, they also highlight Rust's limitations, such as handling memory leaks. While Rust offers robust protections, the authors emphasize that these protections represent only a subset of the broader software security requirements. Their insights are invaluable for understanding both the strengths and limitations of Rust's security model. Wassermann et al. [24] presented a detailed exploration of Rust's security features and potential vulnerabilities. They highlighted issues when design assumptions do not align with real-world data. The authors stress the importance of understanding vulnerabilities from the perspective of Rust program users. They advocate for tools that can analyze these vulnerabilities, even without access to the source code. Discussions also touched upon the maturity of the Rust software ecosystem and its potential impact on future security responses. They suggest that the Rust community could benefit from the Rust Foundation either acting as or establishing a related CVE Numbering Authority (CNA). Their study further enriches the understanding of Rust's security model.

Qin et al. [25] conducted a comprehensive study revealing that unsafe code is widely used in the Rust software they examined. This usage is often motivated by performance optimization and code reuse. They observed that while developers aim to minimize the use of unsafe code, all memory-safety bugs involve it. Most of these bugs also involve safe code,

suggesting that errors can arise when safe code does not account for the implications of associated unsafe code. The researchers identified Rust's 'lifetime' concept, especially when combined with unsafe code, as a frequent source of confusion. This misunderstanding often leads to memory-safety issues. Their findings underscore the importance of fully grasping and correctly implementing Rust's safety mechanisms.

Zheng et al. [26] surveyed the Rust ecosystem for security risks by analyzing a dataset of 433 vulnerabilities, 300 vulnerable code repositories, and 218 vulnerability fix commits over 7 years. They investigated the characteristics of vulnerabilities, vulnerable packages, and the methods used for vulnerability fixes. Key findings reveal that memory safety and concurrency issues constitute two-thirds of the vulnerabilities, with a notable delay (averaging over two years) before vulnerabilities are publicly disclosed. Additionally, they observed an increasing trend in package-level security risks over time despite a decrease in code-level risks since August 2020. Moreover, popular Rust packages tend to have a higher number of vulnerabilities, and vulnerability fixes often involve localized changes. This study contributes to the understanding of security risks in the Rust ecosystem by providing a dataset for future research, summarizing patterns in vulnerability fixes, and discussing implications for securing Rust packages.

Balasubramanian et al.'s research [27] delves deeper into Rust's security aspects by leveraging its linear type system for enhanced safety in system programming. They demonstrate how Rust's safety mechanisms, which incur no runtime cost and eschew garbage collection, are applied to bolster software fault isolation, enforce static information flow control, and facilitate automatic checkpointing. These areas, crucial for security, are shown to benefit from Rust's design, which simplifies the implementation of complex security features. The paper underscores Rust's potential to significantly impact system programming by making high-level security features more attainable without compromising performance. The discussion also acknowledges the learning curve and paradigm shift required to fully utilize Rust's advantages, positioning these as necessary investments for achieving superior safety and security in system programming applications.

A. Security Standards and Guidelines

Various security standards and guidelines can be applied to Rust programming. The International Electrotechnical Commission Technical Report (IEC TR) 24772 [28] standard, "Secure Coding Guidelines Language Independent," provides guidelines suitable for multiple programming languages, including Rust. ISO/IEC 62443 [12], especially sections 4-1 and 4-2, sets the industry standard for secure software development [13]. The Common Weakness Enumeration (CWE) by MITRE [29] offers a unified set of software weaknesses.

The French Government's National Agency for the Security of Information Systems (ANSSI) has published a guide titled "Programming Rules to Develop Secure Applications with Rust" [30], which is a valuable resource for developers.

B. Security Documentation and Tools

Rust's safety guarantees and performance have led to its growing adoption across various domains. Notably, Google has integrated Rust into the Android Open Source Project (AOSP) to mitigate memory safety bugs, a significant source of Android's security vulnerabilities [9]. Updates and discussions about Rust security are frequently shared on blogs, forums, and other platforms.

Several Static Application Security Testing (SAST) tools are available for Rust, such as those listed on the Analysis Tools platform [31]. These tools play a crucial role in the secure software development lifecycle.

Community-driven initiatives like RustSec [10] offer advisories on vulnerabilities in Rust crates (A crate is the smallest amount of code that the Rust compiler considers at a time). Real-time updates from RustSec and other platforms are invaluable for developers to stay updated on potential security issues in Rust packages.

C. Secure Coding Guidelines

The paper "Secure Coding Guidelines - (un)decidability" by Bagnara et al. [32] delves into the challenges of secure coding. It mainly focuses on the undecidability of specific rules, such as "Improper Input Validation". The authors argue that determining adherence to specific secure coding guidelines can be complex due to factors like context.

D. Secure Code Awareness

Secure code awareness is crucial, especially in critical infrastructures. A study by Gasiba et al. [33] explored the factors influencing developers' adherence to secure coding guidelines. While developers showed intent to follow these guidelines, there was a noticeable gap in their practical knowledge. This highlights the need for targeted, secure coding awareness campaigns. The authors introduced a game, the CyberSecurity Challenges, inspired by the Capture The Flag (CTF) genre, as an effective method to raise awareness.

The Sifu platform [34] was developed in line with these challenges. The platform promotes secure coding awareness among developers by combining serious gaming techniques with cybersecurity and secure coding guidelines. It also uses artificial intelligence to offer solution-guiding hints. Sifu's successful deployment in industrial settings showcases its efficacy in enhancing secure coding awareness.

III. METHODOLOGY

Our research methodology, aimed at examining the security in the Rust programming language compared to Java, C, and C++, and its interaction with security assessment tools, was composed as shown in Figure 1.

A. Literature Exploration

Due to the scarcity of academic resources, we commenced with an integrated literature review, primarily focusing on gray literature, such as reports and blog posts. We also conducted an academic literature review using the ACM, IEEE Xplore,

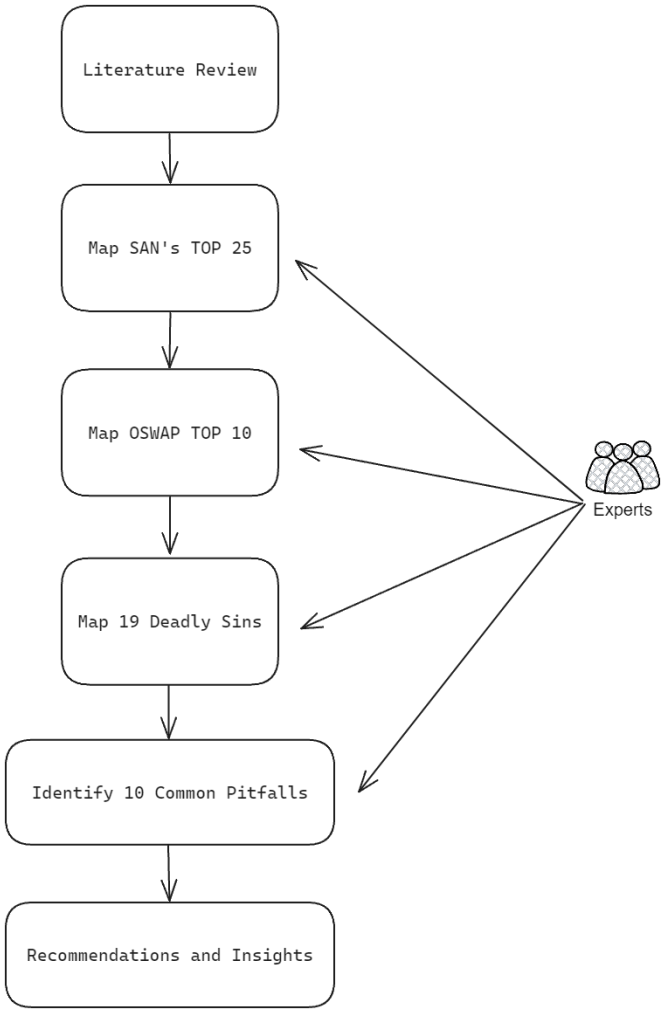

Fig. 1. Methodology

and Google Scholar databases, with search terms including ”Rust Security”, ”Java Security”, ”C Security”, and ”Static Application Security Testing”. The time frame was set from 2010 to early 2024.

B. Interviews with Security Experts

We held discussions with five industry security experts with experience with Rust, Java, C, and security assessment tools. The experts from the industry are consultants with more than ten years of experience and work on the topic of secure software development. Their insights contributed significantly to our understanding and interpretation of the literature. Additionally, we conducted informal surveys with two students who regularly use Rust and contribute to open-source projects developed in the same programming language. The student's background is a master's in computer science with five years of programming experience with Rust. The informal interviews with industry experts and computer science students commenced in August 2023 and lasted about thirty minutes.

C. Mapping to CWE/SANS, OWASP, and 19 Deadly Sins

In this phase, we categorized Rust security issues according to the Common Weakness Enumeration (CWE), SANS Top 25, and OWASP 10 and 19 deadly sins. This step helped in classifying and understanding the security threats relevant to Rust.

D. Analysis with Rust/SAST Tools

A comparative study was undertaken with Rust and Static Application Security Testing (SAST) tools to assess the effectiveness of these tools in identifying Rust's security vulnerabilities.

E. Identifying 10 Common Pitfalls

After our mapping and analysis, we have identified 10 common security pitfalls that developers and stakeholders should be aware of. These pitfalls are not exhaustive but serve as a starting point for stakeholders to consider security as an application-specific issue rather than merely a language-specific concern.

F. Definitions

In our research, we employed three categories to assess the level of security protection against specific issues in Rust: Rare and Difficult (RD), Safeguarded (SG), and Unprotected (UP).

- **Rare and Difficult (RD)**: This category refers to security issues Rust's inbuilt features or mechanisms can fully mitigate or prevent. The language itself provides robust protection against such issues. Security vulnerabilities falling into this category are rare and difficult to spot. They occur infrequently, making it challenging to encounter them. Rust's inherent protections are usually effective in addressing these issues, **unless unsafe blocks are used**. These issues are often not commonly observed and may require specific circumstances or careful analysis, often associated with a Common Vulnerabilities and Exposures (CVE) identifier.
- **Safeguarded (SG)**: Issues falling under this category benefit from protective measures provided by Rust. The programming language offers safeguards to mitigate these issues, reducing their likelihood or impact. However, additional precautions or interventions may be necessary in specific scenarios.
- **Unprotected (UP)**: This category encompasses security issues that the language does not inherently guard against or if the CWE does not apply to the language. The language lacks built-in mechanisms to protect against these issues. Addressing them requires utilizing external libraries or tools or a comprehensive understanding of the language and underlying systems. In cases where a particular CWE is irrelevant to the language, it is also categorized as UP.

We utilized this methodology to evaluate the SANS Top 25, OWASP Top 10, and 19 Deadly Sins of Software Security within the context of Rust. Additionally, we created Proof-of-Concept (PoC) Rust code [35] to validate its feasibility,

containing vulnerabilities for the following weaknesses: Command Injection, Integer Overflow, Resource Leakage, SQL Injection, and Time-of-Check-Time-of-Use (TOCTOU).

IV. RESULTS

A. SANS 25 (2022)

This section presents the findings of our analysis concerning vulnerabilities in Rust, with a particular focus on evaluating vulnerable software based on the SANS Top 25 list. Table I summarizes the protection levels for different CWE vulnerabilities in Rust. These are categorized into three groups: Rare and Difficult (RD), Safeguarded (SG), and Unprotected (UP). It is crucial to note that complete protection is extended to all code that does not use 'unsafe' blocks.

Among the analyzed CWE vulnerabilities, the following are identified as having Full Protection in Rust: CWE-787, CWE-125, CWE-416, CWE-476, CWE-362, and CWE-119. This finding suggests that Rust offers robust protection against these vulnerabilities, thereby minimizing the likelihood of their occurrence in Rust-based software, provided the code does not employ 'unsafe' blocks.

Conversely, several vulnerabilities, including CWE-79, CWE-22, CWE-352, CWE-434, CWE-502, CWE-287, CWE-798, CWE-862, CWE-306, CWE-276, CWE-918, and CWE-611, exhibit No Protection in Rust. This finding implies that Rust lacks built-in mechanisms to prevent or mitigate these vulnerabilities, even when 'unsafe' blocks are not in use. It is vital for developers working with Rust to be cognizant of these vulnerabilities and implement additional security measures to counteract them.

For certain vulnerabilities, such as CWE-79, CWE-20, CWE-78, CWE-190, CWE-77, CWE-400, and CWE-94, Rust provides some protection and safeguards. This result indicates that Rust incorporates certain features or constructs that can help diminish the likelihood of these vulnerabilities. However, additional precautions may still be necessary to mitigate the associated risks fully.

These findings underscore the importance of understanding the vulnerabilities inherent in Rust and implementing suitable security measures. While Rust provides strong protection against specific CWE vulnerabilities, there are areas where additional precautions are necessary. Developers should exercise caution when dealing with vulnerabilities categorized as UnProtected, as these require meticulous attention and specialized security practices.

In addition to analyzing the vulnerabilities in Rust, it is insightful to contrast the protection levels Rust offers with those provided by other prominent programming languages, such as C, C++, and Java. Table II facilitates a side-by-side comparison across these languages. In this table, the protection levels are denoted as follows: Rare and Difficult (RD), Safeguarded (SG), and Unprotected (UP) for C, C++, and Java.

Upon examining Table II, it is evident that C, being an older language, demonstrates fewer protections compared to C++ and Java, especially regarding memory-related vulnerabilities like CWE-787. For instance, C does not provide safeguards for CWE-787 [36], while C++ and Java offer robust protections.

Java, owing to its managed memory model and sandboxed execution environment, shows strong defenses against some vulnerabilities that are particularly problematic in C and C++, such as CWE-416.

Interestingly, for some vulnerabilities like CWE-79 and CWE-22, all three languages - C, C++, and Java - display limited or no protection. This observation accentuates the importance of following secure coding practices irrespective of the language used.

Furthermore, C++ seems to find a middle ground between C and Java regarding protection levels, which could be attributed to its evolution from C and its incorporation of modern language features.

Developers must be cognizant of these variations in protection levels across languages and carefully weigh the security aspects alongside other factors, such as performance and ecosystem, when choosing a language for their projects.

TABLE I
SANS TOP 25 CWE VS. PROTECTION LEVELS IN RUST

CWE ID	Short Description	RD	SG	UP
CWE-787	Out-of-bounds Write	•		
CWE-79	Cross-site Scripting			•
CWE-89	SQL Injection		•	
CWE-20	Improper Input Validation		•	
CWE-125	Out-of-bounds Read	•		
CWE-78	OS Command Injection		•	
CWE-416	Use After Free	•		
CWE-22	Path Traversal			•
CWE-352	Cross-Site Request Forgery			•
CWE-434	Unrestricted Dangerous File Upload			•
CWE-476	NULL Pointer Dereference	•		
CWE-502	Deserialization of Untrusted Data			•
CWE-190	Integer Overflow or Wraparound		•	
CWE-287	Improper Authentication			•
CWE-798	Use of Hard-coded Credentials			•
CWE-862	Missing Authorization			•
CWE-77	Command Injection		•	
CWE-306	Missing Critical Function Authentication			•
CWE-119	Buffer Overflow	•		
CWE-276	Incorrect Default Permissions			•
CWE-918	Server-Side Request Forgery			•
CWE-362	Race Condition	•		
CWE-400	Uncontrolled Resource Consumption		•	
CWE-611	Improper Restriction of XXE			•
CWE-94	Code Injection		•	
		24%	28%	48%

B. OWASP 10

The OWASP Top 10 is a standard awareness document for developers and web application security. It represents a broad consensus about web applications' most critical security risks. The following is an assessment (summarized in Table III) of how the Rust language can offer protection against these vulnerabilities, according to the OWASP standard from 2021:

- **A01-Broken Access Control (SG):** While Rust does not inherently provide web application access control, its strong type system and ownership model can help prevent logical errors that might lead to such vulnerabilities.

TABLE II
SANS TOP 25 CWE VS. PROTECTION LEVELS IN C, C++, AND JAVA

CWE	C			C++			Java		
	RD	SG	UP	RD	SG	UP	RD	SG	UP
CWE-787			•		•		•		
CWE-79			•			•			•
CWE-89			•		•			•	
CWE-20			•			•		•	
CWE-125			•			•	•		
CWE-78			•			•		•	
CWE-416			•		•		•		
CWE-22			•			•			•
CWE-352			•			•			•
CWE-434			•			•			•
CWE-476			•		•		•		
CWE-502			•			•			•
CWE-190			•			•			•
CWE-287			•			•			•
CWE-798			•			•			•
CWE-862			•			•			•
CWE-77			•			•		•	
CWE-306			•			•			•
CWE-119			•		•		•		
CWE-276			•			•			•
CWE-918			•			•			•
CWE-362			•			•		•	
CWE-400			•		•			•	
CWE-611			•			•			•
CWE-94			•			•		•	
	0%	0%	100%	0%	24%	76%	20%	28%	52%

- **A02-Cryptographic Failures (SG):** Although Rust does not provide built-in cryptographic features, it has high-quality cryptographic libraries that can help mitigate these failures to some extent.
- **A03-Injection (SG):** Rust's strong type system and approach to handling strings can help prevent injection attacks. However, poor programming practices may still result in these attacks; see PoC code in [35].
- **A04-Insecure Design (UP):** This vulnerability is more about the design of the application rather than the language itself. While Rust offers memory safety [37], it does not inherently protect against insecure design, which encompasses many issues.
- **A05-Security Misconfiguration (UP):** This vulnerability is more about the application and environment configuration than the language itself.
- **A06-Vulnerable and Outdated Components (SG):** Rust's package manager, Cargo, and its ecosystem can help manage dependencies and their updates.
- **A07-Identification and Authentication Failures (UP):** Rust does not inherently provide user authentication and session management features.
- **A08-Software and Data Integrity Failures (UP):** Rust's ownership model and type system can help ensure data integrity, but it is up to the programmer to leverage these features effectively.
- **A09-Security Logging and Monitoring Failures (UP):** This vulnerability is more about the application's logging and monitoring capabilities than the language itself.
- **A10-Server-Side Request Forgery (SSRF) (UP):** Rust does not inherently protect against SSRF attacks. Programmers must validate and sanitize all URLs and restrict the server's ability to interact only with whitelisted endpoints [38].

We note that in literature, the numbering of the OWASP vulnerabilities can also appear together with the date of the OWASP standard, e.g., A01:2021.

TABLE III
MAPPING OF OWASP TOP 10 FROM 2021 TO RUST PROTECTION LEVELS

OWASP Vulnerability	RD	SG	UP
A01-Broken Access Control		•	
A02-Cryptographic Failures		•	
A03-Injection		•	
A04-Insecure Design			•
A05-Security Misconfiguration			•
A06-Vulnerable and Outdated Components		•	
A07-Identification and Authentication Failures			•
A08-Software and Data Integrity Failures		•	
A09-Security Logging and Monitoring Failures			•
A10-Server-Side Request Forgery			•
	0%	50%	50%

C. 19 Deadly Sins of Software Security

The book "19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them" identifies and guides how to fix 19 common security flaws in software programming (see Table IV for a summary). Rust, a programming language, is designed to prevent some of the most common security vulnerabilities. Below is a brief analysis of how Rust addresses the 19 sins:

TABLE IV
MAPPING OF NINETEEN DEADLY SINS OF SOFTWARE SECURITY TO RUST PROTECTION LEVELS

Security Flaw	RD	SG	UP
Buffer Overflows	•		
Format String Problems		•	
Integer Overflows		•	
SQL Injection		•	
Command Injection		•	
Cross-Site Scripting (XSS)			•
Race Conditions	•		
Error Handling	•		
Poor Logging		•	
Insecure Configuration			•
Weak Cryptography		•	
Weak Random Numbers	•		
Using Known Vulnerable Components		•	
Unvalidated Redirects and Forwards			•
Injection		•	
Insecure Storage			•
Denial of Service		•	
Insecure Third-Party Interfaces			•
Cross-Site Request Forgery (CSRF)			•
	21%	47%	32%

- **Buffer Overflows (RD)**: Rust has built-in protection against buffer overflow errors. It enforces strict bounds

checking, preventing programs from accessing memory they should not.

- **Format String Problems (SG)**: Rust does not support format strings in the same way as languages like C, thereby reducing the risk of this issue. It provides strong protection against format string problems through its type-safe formatting mechanism. The std::fmt module in Rust offers a rich set of formatting capabilities while enforcing compile-time safety.
- **Integer Overflows (SG)**: In Rust, integer overflow is considered a "fail-fast" error. By default, when an integer overflow occurs during an operation, Rust will panic and terminate the program. This behavior helps catch bugs early in development and prevents potential security vulnerabilities. It also offers ways to handle integer overflows gracefully.
- **SQL Injection (SG)**: Rust itself doesn't inherently protect against SQL injection. This protection is usually provided by libraries that parameterize SQL queries, such as rusqlite; see PoC code in [35].
- **Command Injection (SG)**: Rust offers strong protections against command injection vulnerabilities through its string handling and execution mechanisms. The language's emphasis on memory safety and control over system resources helps mitigate the risk of command injection; see PoC code in [35].
- **Cross-Site Scripting (XSS) (UP)**: Rust does not provide inherent protection against XSS. However, web frameworks in Rust, such as Rocket and Actix, have features to mitigate XSS.
- **Race Conditions (RD)**: Rust's ownership model and type system are designed to prevent data races at compile time.
- **Error Handling (RD)**: Rust encourages using the Result type for error handling, which requires explicit handling of errors.
- **Poor Logging (SG)**: Poor logging is more of a design problem than a language issue. Rust offers powerful logging libraries, such as log and env_logger.
- **Insecure Configuration (UP)**: Although Rust's strong typing can catch some configuration errors at compile time, it does not offer direct protections against insecure configurations.
- **Weak Cryptography (SG)**: Rust has libraries that support strong, modern cryptography. However, the correct implementation depends on the developer.
- **Weak Random Numbers (RD)**: Rust's standard library includes a secure random number generator.
- **Using Components with Known Vulnerabilities (SG)**: This is more related to the ecosystem than the language itself. Rust's package manager, Cargo, simplifies updating dependencies.
- **Unvalidated Redirects and Forwards (UP)**: Protection against this is usually provided by web frameworks.
- **Injection (SG)**: Rust's strong typing and absence of eval-like functions lower the risk of code injection.
- **Insecure Storage (UP)**: Not directly related to the language itself.
- **Denial of Service (SG)**: Rust's memory safety and control over low-level details can help build resilient systems, but it does not inherently protect against all types of DoS attacks.
- **Insecure Third-Party Interfaces (UP)**: This issue is usually independent of the programming language.
- **Cross-Site Request Forgery (CSRF) (UP)**: Typically, it is handled by web frameworks rather than the language itself.

In summary, Rust provides strong protections against several of the "19 deadly sins", particularly those related to memory safety and data races. However, some issues, particularly those related to web development or design decisions, are not directly addressed.

D. 10 Common Security Pitfalls in Rust Programming

In our exploration of the security aspects of Rust programming, we have identified several vulnerabilities and potential issues. Beyond the vulnerabilities discussed earlier, it is crucial to highlight general security pitfalls that Rust programmers often encounter. While Rust's safety features significantly reduce certain security risks, awareness and avoidance of common pitfalls are essential for secure coding practices. We outline below 10 common security pitfalls in Rust programming (in Figure 2):

- **Injection Attacks:** Rust's type system can mitigate some risks, but vulnerabilities can arise from improperly handled user input in commands or queries.
- **Integer Overflow and Underflow:** Although Rust provides some level of protection, developers must be vigilant against integer-related vulnerabilities.
- **Request Forgery Attacks:** Rust does not inherently protect against request forgery attacks; developers are responsible for ensuring proper safeguards.
- **Cross-Site Scripting (XSS) Attacks:** In web applications, handling user input safely is vital to preventing XSS vulnerabilities, especially in rendering web pages.
- **Insecure Design:** Despite Rust's advanced features, security vulnerabilities can still result from poor design decisions. It's essential to integrate security considerations into the architectural design. However, these issues are generally independent of Rust and are related to overall security-aware design principles, applicable to any software development.
- **Faulty Access Control:** Inadequate or improperly implemented access control mechanisms can lead to unauthorized access to resources in Rust applications.
- **Logging and Monitoring Failures:** Adequate logging and monitoring are crucial for security, yet often overlooked or poorly implemented in Rust applications, leading to challenges in detecting and responding to security incidents.
- **Security Misconfiguration:** Configuring security settings inadequately in both Rust applications and deployment environments can expose them to risks.

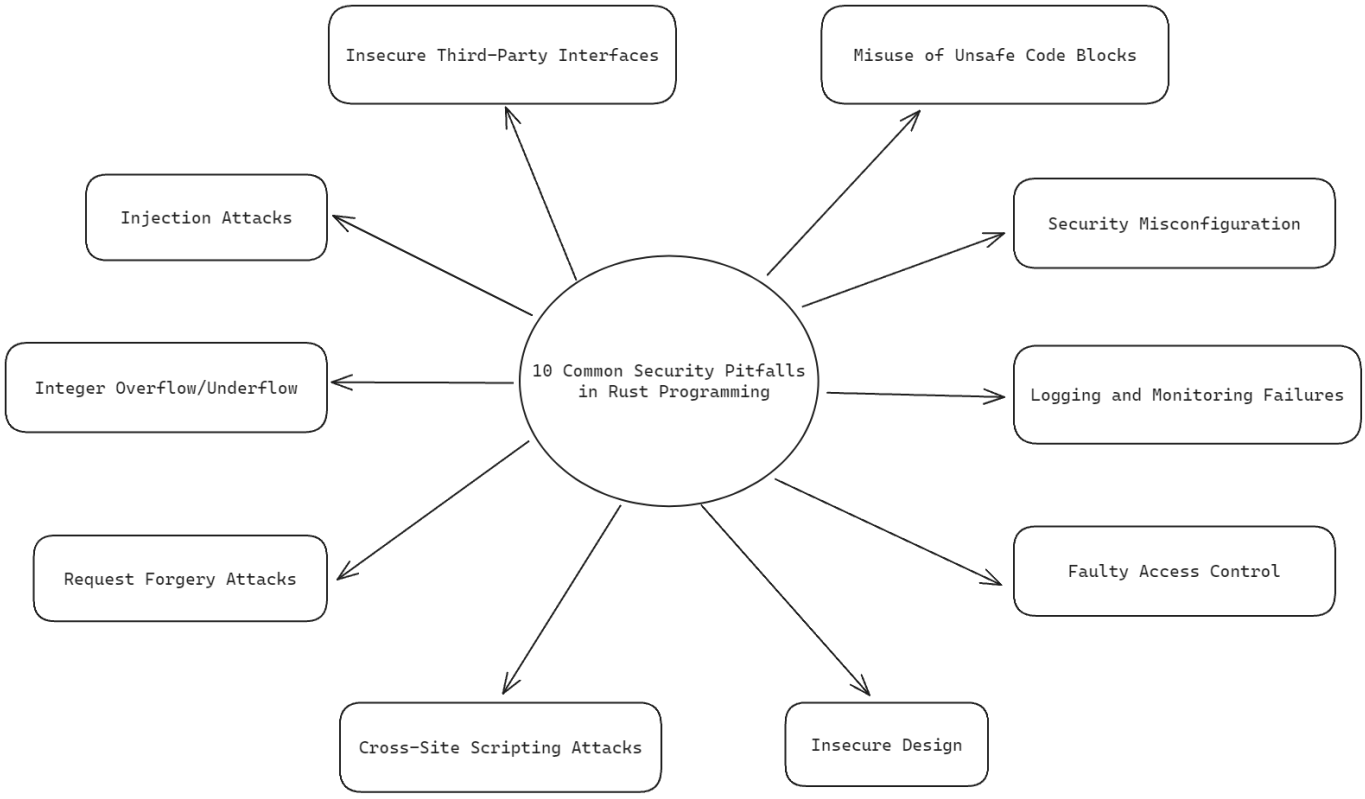

Fig. 2. Ten Common Security Pitfalls in Rust Programming, without Ranking

- **Insecure Third-Party Interfaces:** Relying on third-party libraries or interfaces without proper security vetting can introduce vulnerabilities in Rust applications.
- **Misuse of Unsafe Code Blocks:** Improper use of 'unsafe' code blocks in Rust can compromise the language's inherent safety features, leading to security risks.

This section serves as a guide for common pitfalls and reinforces the importance of comprehensive security practices in Rust development. These insights are intended to support programmers in recognizing and addressing these challenges, thereby enhancing the overall security of software developed in Rust.

1) Injection Attacks: Injection attacks are a prevalent threat in software development, and Rust is not immune to them despite its strong type system and memory safety features. These attacks typically occur when an application unsafely integrates user input into a command or query. While Rust's type system aids in mitigating some risks, vulnerabilities can still arise from improperly validated or sanitized user inputs. This is especially relevant in scenarios involving database queries, command-line arguments, or URL construction, where attackers can exploit unfiltered inputs. Let's examine a specific case of injection attacks: SQL injection.

a) Vulnerable SQL Query in Rust: The following Rust code snippet (Figure 3) demonstrates a vulnerable approach to constructing SQL queries by directly incorporating user input without sanitization:

```
pub fn get_user_by_id(conn: &Connection,
    user_id: &str) -> Result<Vec<User>> {
    // Vulnerable SQL query due to direct
    concatenation of user input
    let query = format!("SELECT id, name, age
    FROM users WHERE id = '{}'", user_id);
    let mut stmt = conn.prepare(&query)?;
    let user_iter = stmt.query_map([], |row| {
        Ok(User {
            id: row.get(0)?,
            name: row.get(1)?,
            age: row.get(2)?,
        })
    })?;
    let mut users = vec![];
    for user in user_iter {
        users.push(user?);
    }
    Ok(users)
}
```

Fig. 3. Vulnerable Code: SQL Injection

This code is prone to SQL injection because it creates a query string by concatenating a user-provided string (‘user_id‘) directly into the SQL command. An attacker could manipulate ‘user_id‘ to alter the structure of the SQL command and execute unintended database operations.

b) Secured SQL Query in Rust: In contrast, the secure version (Figure 4) uses parameterized queries to safely handle user input, effectively preventing SQL injection:

```
fn get_user_by_id_safe(conn: &Connection,
    user_id: &str) -> Result<Vec<User>> {
    let query = "SELECT id, name, age FROM
    users WHERE id = ?";
    let mut stmt = conn.prepare(query)?;
    let user_iter = stmt.query_map([user_id],
    |row| {
        Ok(User {
            id: row.get(0)?,
            name: row.get(1)?,
            age: row.get(2)?,
        })
    })?;

    let mut users = vec![];
    for user in user_iter {
        users.push(user?);
    }

    Ok(users)
}
```

Fig. 4. Safe Code: Prevents SQL Injection

In the secure example, '?' placeholders are used in the SQL query. These placeholders are then filled with the actual 'user_id' in a way that the database engine understands as data, not as part of the SQL command. This approach ensures that even if 'user_id' contains malicious content, it will not be executed as SQL code.

Similarly, for command injections, it is crucial to avoid using user inputs to construct command strings dynamically [39]. Instead, Rust's standard library offers functions to pass arguments to commands in a way that prevents injection, ensuring that inputs are treated as literal text and not as executable code.

This example underscores the importance of carefully handling user inputs in Rust applications. Despite Rust's memory safety features, developers must remain vigilant against vulnerabilities like injection attacks, emphasizing the need for proper input validation and secure coding practices, such as parameterized queries, to maintain application security.

2) Integer Overflow/Underflow: Integer overflow and underflow represent a class of critical vulnerabilities in software applications where arithmetic operations exceed the maximum or minimum limits of the data type being used. Rust provides some level of protection against these vulnerabilities by including checks in the debug mode that cause a panic when an overflow or underflow occurs. However, in release builds, these checks are not enforced by default for performance, which can lead to silent wrapping and potential security risks.

a) Vulnerable Integer Arithmetic in Rust: The following Rust code snippet (in Figure 5) demonstrates an unsafe approach to integer arithmetic, which can lead to overflow or underflow without any warnings or errors in release builds:

```
pub fn withdrawal(balance: u32, amount: u32)
    -> u32 {
    let mut balance = balance;
    balance -= amount;
    balance
}
```

Fig. 5. Vulnerable Code: Integer Overflow/Underflow

In this code, subtracting a larger 'amount' from 'balance' can cause an underflow, which in release mode would wrap around to a very large number due to the unsigned integer type. This could lead to logical errors in the program and potentially severe security vulnerabilities, such as incorrectly authorizing a financial transaction.

b) Secured Integer Arithmetic in Rust: To prevent such issues, Rust offers built-in methods for safe arithmetic that return a 'Result' or an 'Option' type, which can be explicitly handled. Here is a secure version (in Figure 6) of the function that correctly handles underflow using 'checked_sub':

```
pub fn withdrawal(balance: u32, amount: u32)
    -> Result<u32, &'static str> {
    match balance.checked_sub(amount) {
        Some(new_balance) => Ok(new_balance),
        None => Err("Withdrawal not possible:
    insufficient funds"),
    }
}
```

Fig. 6. Safe Code: Prevents Integer Overflow/Underflow

In this secure example, 'checked_sub' is used for subtraction, which returns 'None' if underflow occurs. Using a 'match' statement, the code can handle the underflow case safely, either by returning an error message indicating that the withdrawal is impossible due to insufficient funds or by implementing alternative logic as needed.

This example highlights the need for developers to be aware of the integer arithmetic behavior in their chosen programming language, particularly in a systems language like Rust, often used for low-level tasks. By utilizing Rust's safe arithmetic functions and adequately handling their results, applications can be more robust and secure against integer overflow and underflow vulnerabilities.

3) Request Forgery Attacks: Request Forgery Attacks, such as Cross-Site Request Forgery (CSRF) and Server-Side Request Forgery (SSRF), represent significant security threats to web applications. These attacks exploit a service's trust in the user or the server itself. While Rust's memory safety and concurrency features are commendable, they offer no inherent protection against these web-specific attack vectors.

a) Cross-Site Request Forgery (CSRF): CSRF attacks deceive a web browser into executing an unwanted action on a trusted application where the user is authenticated. In the context of Rust web applications using frameworks like Rocket, such vulnerabilities emerge when state-changing

operations do not require explicit user consent beyond the initial authentication [40].

Consider the example of a Rust application (in Figure 7) with an endpoint to change a user's password, which is vulnerable to CSRF attacks:

```
use rocket::form::Form;

#[derive(FromForm)]
struct PasswordChangeForm {
    new_password: String,
}

#[post("/change_password", data =
    "<password_form>")]
fn change_password(password_form:
    Form<PasswordChangeForm>, user: User) ->
    String {
    // Implement code to change the user's
    password in the database
    // Ensure 'user' is the currently
    authenticated user
    // This code is still CSRF vulnerable
    "Password changed
    successfully".to_string()
}
```

Fig. 7. Vulnerable Code: CSRF

This endpoint is vulnerable because it processes the password change request without verifying the origin of the request, making it susceptible to CSRF attacks. An attacker could craft a malicious website with a form that, when submitted, sends a POST request to this endpoint. If the victim is logged into the Rust application in the same browser, the browser automatically includes the session cookies with the request, leading to an unauthorized password change.

To mitigate CSRF, a token-based strategy is typically employed. As of Rocket version 0.5, automatic CSRF protection was still under discussion, with improvements expected in future versions (See the GitHub issue for the discussion on CSRF protection in Rocket v0.6 [41]).

b) Server-Side Request Forgery (SSRF): SSRF attacks occur when an attacker can induce the server-side application to make requests to arbitrary domains, leading to unauthorized actions or data exposure. Unlike CSRF, SSRF exploits the trust a server has within its own system or between internal services.

In Rust web applications, SSRF can typically occur when user-supplied URLs are used without proper validation to make server-side requests. For instance, fetching a user-specified URL without checking if it points to an internal service or sensitive resource can be exploited.

To prevent SSRF, developers must validate and sanitize all URLs and restrict the server's ability to interact only with whitelisted endpoints. Additionally, following the principle of least privilege when granting network capabilities to the server can mitigate the impact of SSRF attacks.

Request forgery attacks, both CSRF and SSRF, require developers to be proactive in their defense strategies. Employing robust validation, leveraging security features provided by web frameworks, and adhering to security best practices are crucial in protecting Rust web applications from these types of vulnerabilities.

4) Cross-Site Scripting (XSS): Cross-Site Scripting (XSS) attacks are a significant concern in web applications, including those developed in Rust. XSS attacks involve the injection of malicious scripts into webpages viewed by other users, exploiting a user's trust in a particular site.

a) Rust and XSS Vulnerability: Although Rust is known for its robust safety features, it does not inherently protect against XSS attacks [42], [43]. These attacks are primarily concerned with the layer where HTML is generated or manipulated. Rust applications using web frameworks for frontend development, like Rocket, are susceptible to the same XSS vulnerabilities as applications written in other languages.

b) Example of XSS Vulnerability in Rust: Consider a Rust web application that allows users to input data directly displayed on a webpage. An attacker could inject malicious JavaScript code if the application does not properly escape or sanitize the user input. This code could be executed in the browsers of other site users, leading to data theft, session hijacking, or other malicious activities.

c) Protecting Against XSS in Rust: To mitigate XSS risks in Rust applications, developers need to:

- **Escape User Input:** Ensure that user inputs are correctly escaped before rendering on web pages. This prevents malicious scripts from being executed [44].
- **Use Safe Frameworks and Libraries:** Employ frameworks and libraries that automatically handle escaping. For instance, the Ammonia crate in Rust sanitizes HTML to prevent XSS by filtering out harmful tags and attributes (Sometimes even crates like Ammonia have XSS vulnerabilities [43], so developers should always keep themselves informed about such vulnerabilities).
- **Content Security Policy (CSP):** Implement a strong CSP to reduce the severity of any XSS vulnerabilities by restricting where scripts can be loaded from and executed [44].
- **Validate and Sanitize Input:** Rigorously validate and sanitize all user input, especially data that will be included in HTML output.
- **Regular Security Audits:** Conduct regular security reviews and testing, including dynamic application security testing (DAST) and penetration testing, to identify and fix XSS vulnerabilities [44].

While Rust provides significant advantages regarding memory safety, developers must still be diligent in protecting against XSS and other web-based vulnerabilities. Implementing best practices for input handling and utilizing security-focused libraries and frameworks are essential steps in creating secure Rust web applications.

5) Insecure Design: Insecure design in software development refers to flaws that arise not from specific coding errors but from fundamental issues in the software's architecture and design decisions. Even in a language like Rust, known for

its emphasis on safety and security, an application's overall security is significantly influenced by its design. We explore common design pitfalls in Rust programming and recommend strategies to foster secure software design.

a) Impact of Insecure Design: While Rust provides strong guarantees against memory safety issues, it does not inherently address higher-level design vulnerabilities such as authentication flaws, inadequate data protection, or insecure communication protocols. These vulnerabilities are often the result of oversight during the design phase and can lead to significant security risks, including data breaches, unauthorized access, and system compromises.

b) Common Design Flaws in Rust Applications:

- **Insufficient Authentication and Authorization:** Overlooking robust authentication and authorization mechanisms can lead to unauthorized access. Rust applications, especially those interfacing with web services, must implement strong authentication protocols and ensure user privileges are correctly managed.
- **Lack of Data Encryption:** Failing to encrypt sensitive data at rest and in transit can expose it to interception and misuse. Rust applications handling confidential information should utilize strong encryption algorithms and libraries to secure data.
- **Ignoring Secure Communication Protocols:** Neglecting to use secure communication channels like HTTPS can make Rust applications vulnerable to man-in-the-middle attacks. Ensuring encrypted communication is critical, especially for web-based applications.

c) Best Practices for Secure Design: To mitigate the risks associated with insecure design, the following best practices are recommended:

- **Threat Modeling:** Early in the design process, conduct threat modeling to identify potential security threats. This proactive approach helps in designing systems that are resilient against identified risks [45], [46].
- **Principle of Least Privilege:** Design systems where components operate with the minimum privileges necessary. This reduces the impact of a potential compromise [47].
- **Secure Defaults:** Ensure that the application's default configuration is secure. This includes settings for user access, data processing, and communication protocols [46].
- **Regular Security Audits:** Conduct regular security reviews and audits of the design to identify and address new and evolving security threats [46].
- **Staying Informed:** Keep abreast of the latest security trends and best practices in software design. Applying up-to-date knowledge can significantly enhance the security posture of Rust applications.

In Rust development, as in any software development endeavor, secure design is just as crucial as secure coding. An application's architecture and design decisions lay the foundation for its overall security. By adhering to best practices in secure design and being mindful of common design pitfalls, Rust developers can significantly reduce the risk of security vulnerabilities in their applications.

6) Faulty Access Control: Faulty access control is a critical security issue that can lead to unauthorized access and data breaches. In the context of Rust programming, while the language offers strong memory safety features, access control largely depends on the application's design and the use of libraries. This section discusses the common challenges and best practices in implementing robust access control in Rust applications.

a) Challenges in Access Control: Access control mechanisms are essential for defining and enforcing who can access what resources in an application. In Rust, the challenges in implementing access control often stem from:

- **Complex User Permissions:** Managing complex user permissions and roles can be challenging, especially in applications with multiple user levels and diverse access needs.
- **Dependency on External Libraries:** Rust's standard library does not provide specific features for access control, leading developers to rely on external libraries, which might vary in their security robustness.
- **Inadequate Session Management:** Implementing secure session management is crucial for web applications. Faulty session management can lead to vulnerabilities like session hijacking and fixation.

b) Best Practices for Access Control: To ensure effective access control in Rust applications, consider the following best practices:

- **Role-Based Access Control (RBAC):** Implement RBAC to manage user permissions efficiently. RBAC allows for grouping permissions into roles, which can be assigned to users, simplifying the management of user privileges.
- **Use of Vetted Libraries:** When relying on external libraries for access control features, choose well-vetted libraries with a strong security track record. Regularly update these libraries to incorporate security patches.
- **Secure Authentication and Session Management:** Implement strong authentication mechanisms and ensure that session management is secure. This includes using secure tokens, implementing session timeouts, and protecting against common session attacks.
- **Regular Access Reviews:** Regularly review and update access control policies to ensure they align with the current organizational structure and user roles.
- **Audit and Logging:** Maintain comprehensive audit logs for access control events. Monitoring and analyzing these logs can help detect unauthorized access attempts and improve the overall security posture.

Faulty access control can have severe implications for the security of a Rust application. While Rust provides the tools for building safe and concurrent applications, access control relies more on the application's design and the secure implementation of libraries and frameworks. By adopting

robust access control practices and staying vigilant in their application, developers can significantly enhance the security of their Rust applications.

7) Logging and Monitoring Failures: Effective logging and monitoring are crucial for the security and stability of any software application, including those developed in Rust. While Rust's language features promote safety and concurrency, they do not inherently provide solutions for logging and monitoring. This section addresses common pitfalls in logging and monitoring Rust applications and suggests best practices to mitigate these issues.

a) Importance of Logging and Monitoring: Logging and monitoring are pivotal in detecting, diagnosing, and responding to security incidents and system failures. In Rust applications, ineffective logging and monitoring can lead to:

- **Inadequate Detection of Security Incidents:** Without proper logging, malicious activities or security breaches may go unnoticed, increasing the risk of damage.
- **Difficulty in Troubleshooting and Debugging:** Insufficient logging can hinder the identification and resolution of bugs or performance issues, affecting the reliability and efficiency of the application.
- **Compliance Issues:** Failure to maintain adequate logs can lead to non-compliance with regulatory requirements, especially in industries where logging is mandated for audit trails.

b) Best Practices for Logging and Monitoring: Implementing effective logging and monitoring in Rust applications involves several best practices:

- **Comprehensive Logging Strategy:** Develop a logging strategy that covers what to log, at what level, and how to securely store and manage logs. Ensure that logs capture essential information for security and operational insights.
- **Use of Robust Logging Frameworks:** Utilize mature and well-supported logging frameworks in Rust, such as `log` and `env_logger`, which provide flexibility and ease of integration.
- **Real-time Monitoring and Alerting:** Implement real-time monitoring tools to promptly detect and alert on abnormal activities or performance issues.
- **Log Analysis and Correlation:** Regularly analyze logs to identify patterns or anomalies. Correlate logs from different sources to gain a comprehensive understanding of events.
- **Secure and Compliant Log Management:** Ensure logs are stored securely, with access controls in place. Logs should be managed in compliance with data protection regulations.

Effective logging and monitoring are vital for maintaining the security and integrity of Rust applications. By implementing a robust logging and monitoring strategy and utilizing appropriate tools and practices, developers can significantly enhance their ability to detect, diagnose, and respond to Rust application issues, reinforcing their overall security and reliability.

8) Security Misconfiguration: Security misconfiguration is one of the most common vulnerabilities in software applications, arising from improper setup or default configurations that are not secure. In Rust applications, as with any other technology, attention to configuration details is crucial for ensuring system security. This section highlights the typical areas where security misconfiguration can occur in Rust applications and provides guidelines to prevent such vulnerabilities.

a) Typical Areas of Misconfiguration: Security misconfigurations in Rust applications can manifest in various ways:

- **Default Settings:** Leaving default settings unchanged, especially those related to security, can expose applications to known vulnerabilities [48].
- **Insecure Database Configurations:** Inadequately secured database connections or default credentials can lead to unauthorized access [48].
- **Improper File Permissions:** Incorrect file and directory permissions can give attackers access to sensitive data or system files [48].
- **Exposed Sensitive Information:** Exposing sensitive information like debug details, stack traces, or cryptographic keys through error messages or logs [48].
- **Lack of Security Features in External Libraries:** Using external libraries without properly configuring their security features.

b) Best Practices to Prevent Security Misconfiguration: Implementing the following best practices can significantly reduce the risk of security misconfiguration in Rust applications:

- **Regular Configuration Reviews:** Conduct regular reviews of application configurations, particularly after updates or changes, to ensure security settings are appropriate and up to date.
- **Minimal Necessary Permissions:** Apply the principle of least privilege to file, database, and network permissions. Only grant access levels necessary for the operation.
- **Secure Default Settings:** Customize default settings to enforce security, including turning off unnecessary features and services.
- **Manage Sensitive Information:** Ensure sensitive information like keys, credentials, and personal data is securely managed and not exposed in logs or error messages.
- **Update and Patch Libraries:** Regularly update external libraries to incorporate security patches and review their configurations for security implications.
- **Security Hardening Guides and Checklists:** Utilize security hardening guides and checklists to systematically address potential misconfigurations.

Security misconfiguration can lead to severe vulnerabilities in Rust applications. By being vigilant about configuration details, regularly reviewing and updating settings, and following best practices for security management, developers can significantly enhance the security posture of their applications, mitigating the risks associated with misconfiguration.

9) Misuse of Unsafe Code: The Rust programming language is lauded for its emphasis on safety, particularly memory safety. However, Rust also provides an ‘unsafe‘ keyword, allowing developers to opt out of some of these safety guarantees for various reasons [49], such as interfacing with low-level system components or optimizing performance [50]. While powerful, the misuse of ‘unsafe‘ code can introduce significant security vulnerabilities. This section discusses the responsible use of ‘unsafe‘ code in Rust and strategies to minimize its risks.

a) Risks Associated with Unsafe Code in Rust: ‘Unsafe‘ code in Rust bypasses the compiler’s safety checks, which can lead to several risks:

- **Memory Safety Violations:** ‘Unsafe‘ code can lead to issues like dereferencing null or dangling pointers, leading to undefined behavior or security vulnerabilities such as buffer overflows [50], [51].
- **Concurrency Issues:** Incorrect handling of concurrent operations in ‘unsafe‘ code can result in data races and undefined behavior [51].
- **Violations of Rust’s Ownership Model:** ‘Unsafe‘ code can violate Rust’s ownership rules, potentially leading to memory leaks or double-free errors [50], [51].

Consider a scenario depicted in Figure 8, where an unsafe block executes low-level operations, such as opening a file through direct system calls. In these instances, the safety mechanisms of Rust, designed to prevent memory and resource leaks, are circumvented. Should the programmer fail to explicitly close the file descriptor acquired via these operations, it may result in resource leakage. This differs from memory leaks, which involve un-freed allocated memory. Here, resource leakage denotes the depletion of available file descriptors, a limited system resource. Such oversight can lead to the application’s inability to open new files or sockets upon reaching the open file descriptor limit, potentially causing broader system issues if not adequately addressed. Consequently, ensuring that every file descriptor opened within an unsafe block is properly closed is imperative, thereby maintaining system stability and averting resource leakage.

b) Best Practices for Using Unsafe Code in Rust: To mitigate the risks associated with ‘unsafe‘ code, consider the following best practices:

- **Minimize Use of Unsafe Code:** Limit the use of ‘unsafe‘ code to situations where it is necessary, such as interfacing with hardware or legacy C code [51].
- **Isolate Unsafe Code:** Encapsulate ‘unsafe‘ code in small, well-defined modules or functions. This isolation makes it easier to audit and test the unsafe portions of your codebase.
- **Document Unsafe Code:** Clearly document the reasoning behind the use of ‘unsafe‘ code and the precautions taken to uphold safety guarantees.
- **Peer Review:** Unsafe code should undergo rigorous peer review by experienced Rust developers, focusing on the necessity and safety of the ‘unsafe‘ operations.

```
use libc::{c_char, c_int, open, read,
    O_RDONLY};
use std::{ffi::CString, thread,
    time::Duration};

fn read_file(filename: &str) ->
    Result<String, String> {
    let c_filename =
    CString::new(filename).expect("CString::new
    failed");
    let fd: c_int = unsafe {
    open(c_filename.as_ptr() as *const
    c_char, O_RDONLY) };

    if fd < 0 {
        return Err("Unable to open
    file".to_string());
    }

    let mut buffer = vec![0u8; 1024];
    let bytes_read = unsafe { read(fd,
    buffer.as_mut_ptr() as *mut libc::c_void,
    buffer.len()) };

    if bytes_read < 0 {
        return Err("Unable to read
    file".to_string());
    }

    // Remove extra null bytes from the buffer
    buffer.resize(bytes_read as usize, 0);

    // Failing to close the file descriptor
    results in resource leakage.
    // unsafe { close(fd); }

Ok(String::from_utf8_lossy(&buffer).into_owned())
}

fn main() {
    let filename = "test.txt";
    loop {
        println!("Reading file...");
        match read_file(filename) {
            Ok(contents) => println!("File
    contents: {}", contents),
            Err(err) => println!("Error: {}",
    err),
        }

    thread::sleep(Duration::from_millis(50));
    }
}
```

Fig. 8. Vulnerable Code: Resource Leak with Unsafe Rust

- **Automated Testing:** Implement comprehensive automated testing, including fuzz testing, to uncover potential issues in ‘unsafe‘ code [14].
- **Stay Informed:** Keep up-to-date with best practices and guidelines from the Rust community regarding the use of ‘unsafe‘ code.

The use of ‘unsafe‘ code in Rust, while sometimes nec-

essary, should be approached with caution. By minimizing its use, isolating it, documenting it, and rigorously testing it, developers can mitigate the risks associated with bypassing the compiler's safety checks. Responsible use of 'unsafe' code is essential for maintaining the security and reliability of Rust applications.

10) Insecure Third-Party Interfaces: In today's software development ecosystem, reliance on third-party interfaces and libraries is commonplace. While this approach boosts efficiency and productivity, it also introduces potential security risks, especially if these third-party components are insecure or misconfigured. Rust applications are no exception to this, and developers must be vigilant about the third-party interfaces they integrate. This section explores the challenges and best practices related to using third-party interfaces in Rust applications.

a) Challenges with Third-Party Interfaces: Integrating third-party interfaces in Rust applications can present several challenges:

- **Dependency Vulnerabilities:** Libraries and interfaces may contain vulnerabilities that can be exploited, compromising the security of the entire application.
- **Lack of Control:** Using third-party components often means relinquishing control over certain aspects of the application, which might lead to security lapses if these components are not adequately maintained.
- **Incompatibilities:** Incompatibilities between different libraries or between libraries and the Rust runtime can lead to unexpected behavior and potential security issues.
- **Outdated Components:** Using outdated versions of libraries and interfaces can expose applications to known vulnerabilities that have been fixed in later versions.

b) Best Practices for Secure Use of Third-Party Interfaces: To safely integrate third-party interfaces in Rust applications, consider the following best practices:

- **Vet Third-Party Libraries:** Before integrating a third-party library, vet it for security. Check its reputation, maintenance history, and community feedback.
- **Keep Dependencies Updated:** Regularly update third-party libraries to the latest versions to ensure you have the most recent security patches and features.
- **Monitor for Vulnerabilities:** Use tools to monitor dependencies for known vulnerabilities. Automated vulnerability scanning tools can be integrated into the development workflow.
- **Understand the Code:** As much as possible, understand the code of third-party libraries, at least those parts you integrate into your application. This understanding can be crucial for identifying potential security risks [52].
- **Limit Dependencies:** Minimize the number of third-party dependencies. The more dependencies your project has, the larger the attack surface.
- **Isolate Critical Components:** Isolate critical components of your application from third-party code. This isolation can prevent cascading failures or security breaches from affecting core functionalities [52].

While third-party interfaces are invaluable in modern software development, their integration must be cautiously approached, especially in the context of Rust applications. By carefully selecting, monitoring, and managing these third-party components, developers can mitigate potential security risks and maintain the integrity and security of their Rust applications.

In the following sections, we will delve deeper into the analysis of past vulnerabilities in the Rust language and its ecosystem and shed light on the time taken to address these vulnerabilities and the current open issues in the Rust security landscape. This comprehensive analysis aims to provide a better understanding of the vulnerabilities in Rust and guide developers and researchers in effectively addressing security concerns in Rust-based software.

E. CVEs Addressed by Rust Security Advisory

A quick search on CVE Mitre with the keyword "Rust" returns over 400 vulnerabilities at the time of writing. Various researchers have analyzed the CVEs, and the Rust community actively fix them once discovered [10], [53]. However, Rust's security advisory only addresses some of these vulnerabilities: CVE-2021-42574 [54], CVE-2022-21658 [55], CVE-2022-24713 [56], CVE-2022-36113 [57], CVE-2022-36114 [57], CVE-2022-46176 [58], CVE-2023-38497 [59] and CVE-2024-24576 [60].

One of the CVEs acknowledged by the Rust security advisory on their blog is CVE-2022-46176 [58]. This vulnerability, found in Cargo's Rust package manager, could allow for man-in-the-middle (MITM) attacks due to a lack of SSH host key verification when cloning indexes and dependencies via SSH. All Rust versions containing Cargo before 1.66.1 are vulnerable. Rust version 1.66.1 was released to mitigate this, which checks the SSH host key and aborts the connection if the server's public key is not already trusted.

F. Comparison of Rust Static Analysis Tools with Python, Java, and C++

Rust has been gaining traction due to its focus on safety and performance. As a young language, Rust's ecosystem of static analysis tools is still in rapid development. The primary tool for static analysis in Rust is the Rust compiler, which includes a robust type system and borrow checker that prevents many bugs at compile time. Moreover, tools like Clippy [61] and Mirchecker [62] provide lints to catch common mistakes and improve Rust code.

In contrast, languages like Python, Java, and C++ have been around for a considerable time and have a mature set of static analysis tools. Python, a dynamically typed language, relies on tools like PyLint, PyFlakes, and Bandit for static analysis. With its static type system, Java uses tools like FindBugs, PMD, and Checkstyle. C++, known for its complexity and flexibility, employs tools like cppcheck and Clang Static Analyzer.

While each language has its unique set of static analysis tools, the effectiveness of these tools can vary based on the

language's features and characteristics. The rapidly evolving Rust ecosystem is a testament to the language's growing popularity and commitment to safety and performance. On the other hand, the mature toolsets of Python, Java, and C++ provide robust support for detecting potential bugs and improving code quality, backed by years of development and refinement.

V. DISCUSSION

In this study, we have explored the security implications of using the Rust programming language, which is gaining traction in the software industry due to its claims of safety and security. Our findings indicate that while Rust offers certain security advantages, it is not immune to vulnerabilities, and there are areas where it falls short compared to other, more mature languages.

A. Rust Security Model and Its Attributes

The Rust programming language incorporates a comprehensive security model to facilitate safe systems programming. The attributes of this model are integral to its capability to minimize common security vulnerabilities. These attributes of Rust are also the reason why many people love the language so much [63]. Figure 9 provides a visual representation of these attributes, each contributing to the overall robustness of the language's approach to security.

1) Key Attributes of the Rust Security Model:

a) Ownership and Borrowing: Central to Rust's memory safety guarantees are the principles of ownership and borrowing. These mechanisms ensure that memory is properly allocated and deallocated, preventing common vulnerabilities such as buffer overflows and dangling pointers.

b) Lifetime Tracking: Rust's compiler enforces lifetime annotations, which specify the scope of validity for references. This prevents memory leaks and unauthorized memory access, which are typical in systems programming.

c) Public/Private Module Management: By allowing developers to define public or private modules, Rust ensures encapsulation and control over data exposure, mitigating the risks associated with unauthorized access (by default modules are private).

d) Safe and Unsafe Code: While Rust defaults to safe code, it provides the 'unsafe' keyword to perform certain low-level operations. This dichotomy allows for flexibility where necessary yet maintains strict safety checks by default.

e) Memory Safety Without Garbage Collection: Rust provides memory safety guarantees without a garbage collector, offering deterministic performance critical in systems programming.

f) Zero-Cost Abstractions: Rust's zero-cost abstractions allow developers to write high-level abstractions without performance penalties, often associated with high-level languages.

g) Type System and Concurrency: The language's type system and concurrency model prevent data races and ensure safe concurrent programming, a common source of vulnerabilities in multi-threaded applications.

h) Compile-Time Memory Management: Rust's ability to manage memory at compile time prevents a class of bugs that can be exploited at runtime, enhancing the security and performance of applications.

i) Error Handling: Rust uses the 'Result' and 'Option' types for error handling, which prevents crashes due to unhandled errors and exceptions, enhancing the reliability of the application.

j) Cargo Package Manager and Dependency Management: Rust's package manager, Cargo, assists in managing dependencies securely and facilitates easy maintenance and updates, ensuring that security updates are seamlessly integrated into the development process.

B. Mapping of SANS TOP 25, OWASP TOP 10 and 19 Deadly Sins

Our research has shown that writing vulnerable software in Rust is possible. This finding is essential, as it challenges the perception that Rust is inherently secure. While Rust's design does make some types of vulnerabilities harder to introduce, it is not a panacea. Other security aspects are as problematic in Rust as in any other language. This point underscores the fact that while language choice can influence the security of a software system, it is not the only factor. Good security practices are essential, regardless of the language used.

Some vulnerabilities are hard or impossible to solve through an improved programming language as these belong to a "non-decidable" category. Therefore, writing a compiler or defining a programming language that identifies and eliminates such problems is impossible. However, we have observed that Rust does offer improvements over other languages in handling these issues, which is a positive sign.

One of the challenges we encountered in our research is the relative immaturity of Rust compared to other languages. There are fewer studies on Rust security, and the tools and support for secure development are not as robust. For example, SonarQube [64], a popular tool for static analysis of code to detect bugs, code smells, and security vulnerabilities, does not currently support Rust. This lack of tooling can significantly impede Rust's adoption in an industrial context, where such tools are critical for finding vulnerabilities and passing cybersecurity certifications.

Our discussions with industry experts found that Rust's high learning curve is another potential barrier to its adoption. More investigation is needed to understand the security consequences of this compared to other languages that might be easier to learn. The lack of a "competent" workforce skilled in Rust is another challenge that needs to be addressed.

In our analysis of the SANS Top 25, Rust provides inherent protections against 24% of the vulnerabilities, some safeguards against 28% of vulnerabilities, and does not offer protection or does not apply to 48% of the vulnerabilities. We made notable observations when comparing Rust with other programming languages like C, C++, and Java. C does not offer any inherent protections against the vulnerabilities listed in the SANS Top 25, as it was designed to be minimal and efficient.

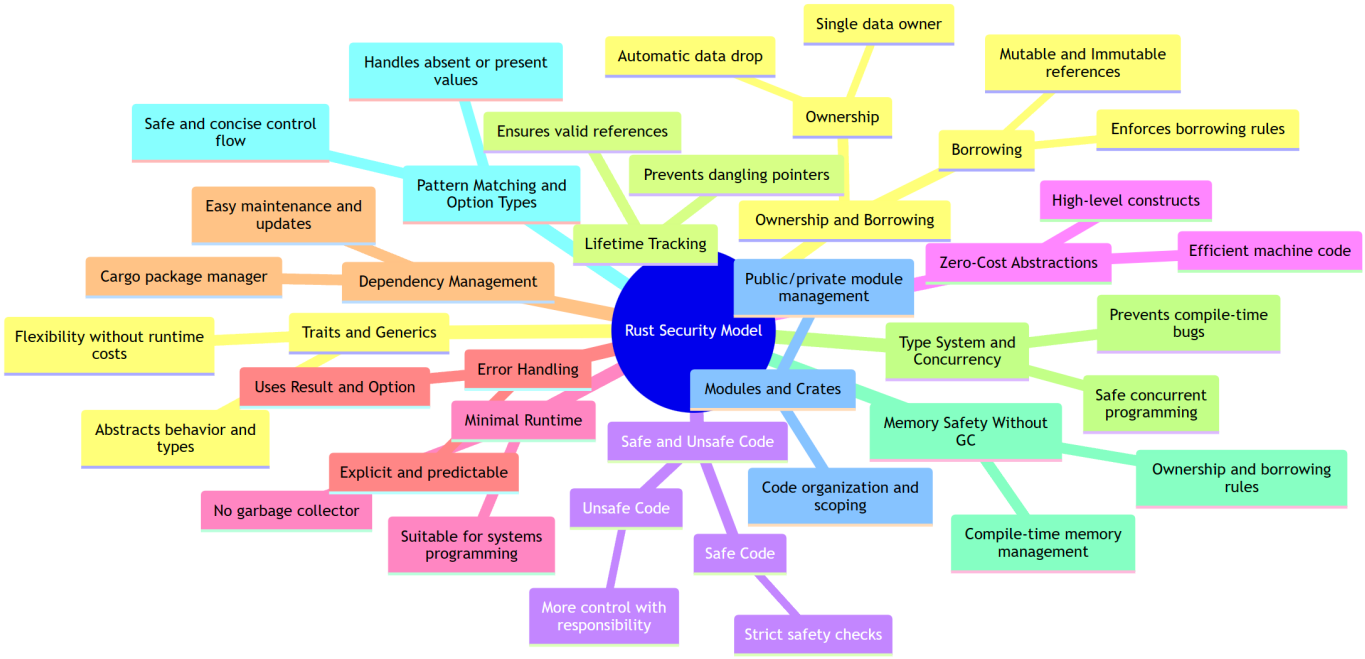

Fig. 9. Rust Security Model

C++, on the other hand, provides safeguards against particular vulnerabilities, such as CWE-787 and CWE-15. Examples of language features that can protect against these vulnerabilities include the C++ Standard Template Library (STL) and other features. Nevertheless, the C++ programming language does not inherently protect against them. In our study, we observe that C++ safeguards against only 24% of the vulnerabilities in the SANS Top 25. However, Java utilizes a garbage collector that inherently protects against memory-related issues. This feature puts Java closer to Rust in terms of protection [65].

Our analysis of the OWASP findings revealed that not a single finding is of the type RD, which is to be expected, as Rust is more a system-level programming language rather than a programming language for web technologies. Compared to C, C++, and Java, which are widely used in the industry, Rust shows promise but has limitations.

Our analysis of the 19 Deadly Sins showed that Rust provides inherent protections against 21% of these sins, offers safeguards for 47% of them, and leaves 32% of the sins unprotected.

We do not expect any current or future programming language to be able to cover 100% of the vulnerabilities, as many coding guidelines in CWE are non-decidable. However, our work shows that Rust does a commendable job addressing many CWE guidelines.

Our inspiration to use a three-point scale (RD, SG, and UP) in our analysis is based on the work by Jacoby (1971) [66], who argued that ”Three-point Likert scales are good enough.”

In addition to our Rust security vulnerabilities analysis, we have also delineated 10 common security pitfalls that developers should be vigilant about. This compilation is not exhaustive but serves as a crucial starting point for Rust developers to cultivate a security-first mindset. By highlighting these pitfalls—ranging from Injection Attacks to the Misuse of Unsafe Code—we aim to underscore the multifaceted nature of security in Rust programming.

These pitfalls were selected based on their prevalence and potential impact on the security of Rust applications. Pitfalls are accompanied by examples and best practices, offering developers concrete strategies to avoid or mitigate these risks. This proactive approach is essential in a landscape where security threats are continually evolving, and developer awareness can significantly influence the resilience of software systems.

Furthermore, discussing these pitfalls complements our broader research findings, providing a comprehensive overview of the security considerations specific to Rust. It acknowledges that while Rust’s design offers significant safety features, security is a broader concern that extends beyond the language’s inherent mechanisms. Developers must remain cognizant of the various ways vulnerabilities can manifest, whether through logical errors, misconfigurations, or the integration of insecure third-party components.

Our exploration of common security pitfalls in Rust programming serves as both a cautionary tale and a roadmap for secure development practices. As the Rust ecosystem continues to grow and evolve, so will the challenges and strategies for ensuring the security of Rust applications. We hope this guide will be a valuable resource for developers, encouraging a holistic approach to security that integrates seamlessly with Rust’s safety features.

The authors consider the present work essential as Rust's usage for software development continues to grow. Without awareness of potential vulnerabilities, we risk replacing one problem with another. It is crucial to emphasize the security limitations of Rust early on rather than treating security as an add-on feature. Security should be prioritized from the inception of every project. Furthermore, due to Rust being a relatively new language, standardized testing tools for assessing compliance with ISO/IEC security standards are not yet available, or very few. This lack of tools makes it challenging to introduce Rust into the industry.

The present work does not focus on finding novel software weaknesses specific to the Rust programming language but rather on comparing well-known vulnerabilities, e.g., as present in secure programming standards, and their relation to the Rust programming language. Additional investigation is needed to understand potential vulnerabilities when developing software in Rust which are caused by the language itself.

In conclusion, our work contributes to scientific knowledge and industry practice by shedding light on the security implications of using Rust. While Rust is rising in significance and the industry is starting to adopt it, there is a lack of studies on its security aspects. Our work closes this gap and shows that while it is still possible to write vulnerabilities in Rust, some problems are well-considered. As Rust continues to grow in popularity, we hope our findings will help guide its development in a direction that prioritizes security and that our work will serve as a foundation for further research in this area.

While the interviews carried out in the present work include a limited number of participants, the results of the present work are validated. The authors did not only confirm some vulnerabilities with proof-of-concept code but also conducted interviews with highly experienced security experts. Nevertheless, the mapping to protection levels, while dependent on the authors' and interviewees' experience, can also change in future releases of the Rust programming language.

VI. CONCLUSION AND FUTURE WORK

Our research provided valuable insights into the security implications of the Rust programming language. While Rust has significantly enhanced software security, we have demonstrated that it is not immune to vulnerabilities. Our findings challenge the notion that Rust is inherently secure and highlight the need for robust security practices, regardless of the language used.

Our study has also shed light on the challenges associated with Rust's relative immaturity compared to other, more established languages. The lack of comprehensive studies on Rust security, the absence of robust tooling for secure development, and the high learning curve associated with Rust are all areas that require attention. Furthermore, the shortage of a skilled workforce in Rust is a significant barrier that needs to be addressed to facilitate its broader adoption in the industry.

Despite these challenges, Rust shows promise. Its design makes specific vulnerabilities harder to introduce and offers improvements over other languages in handling "non-decidable" problems. As Rust continues gaining traction in the software industry, it is crucial to investigate its security implications and develop tools and practices to mitigate potential vulnerabilities.

As the following steps, there are several avenues for future work. One of the critical areas is the development of tools to support secure development in Rust. These tools include static application security testing tools like SonarQube, which are critical for finding vulnerabilities and passing cybersecurity certifications. Another area of focus is the development of comprehensive training programs to lower Rust's learning curve and build a competent workforce skilled in Rust. Further research should also focus on studying real-world examples of security concerns in Rust applications to strengthen the points made by the authors. Additionally, providing suggestions on what Rust needs to improve, specifically in terms of the 10 common security pitfalls identified, will be essential. Comparing Rust with the Go programming language, particularly regarding inherent security considerations, will also provide valuable insights.

As more software is developed in Rust, it is crucial to maintain a sense of urgency in highlighting its security shortcomings. Security should not be an afterthought but should be integrated from the beginning of every project. We hope our work will contribute to developing safer and more secure software systems.

REFERENCES

[1] T. E. Gasiba and S. Amburi, "I Think This is the Beginning of a Beautiful Friendship - On the Rust Programming Language and Secure Software Development in the Industry," in *Proceedings of the Eighth International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2023)*, IARIA. ThinkMind, September 2023, pp. 19–26. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=cyber_2023_1_40_80031

[2] Yalantis, "Rust Market Overview," 2023, accessed: July 16, 2023. [Online]. Available: https://yalantis.com/blog/rust-market-overview/

[3] E. D. C. Garcia, "Rust Makes Us Better Programmers," 2023, accessed: July 16, 2023. [Online]. Available: https://thenewstack.io/rust-makes-us-better-programmers/

[4] S. O. Ryan Donovan, "Why the developers who use Rust love it so much," Jun 2020, accessed: July 16, 2023. [Online]. Available: https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/

[5] S. Overflow, "Stack Overflow Developer Survey 2023," https://survey.stackoverflow.co/2023/#section-admired-and-desired-programming-scripting-and-markup-languages, 2023, accessed: July 16, 2023.

[6] J. Barron, "Rust's Addition to the Linux Kernel Seen as 'Enormous Vote of Confidence' in the Language," *SD Times*, Nov. 2022, accessed: July 16, 2023.

[7] Writing Linux Kernel Modules in Rust. [Online]. Available: https://www.linuxfoundation.org/webinars/writing-linux-kernel-modules-in-rust

[8] Office of the National Cyber Director, "Title of the report," White House, Washington, D.C., Tech. Rep., 02 2024. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

[9] G. S. Team, "Memory-safe languages in Android 13," 2022, accessed: July 16, 2023. [Online]. Available: https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

[10] "RustSec Advisory Database," 2023, accessed: July 16, 2023. [Online]. Available: https://rustsec.org/advisories/

[11] National Security Agency, "U.s. and international partners issue recommendations to secure software products," Press Release, 12 2023. [Online]. Available: https://www.nsa.gov/Press-Room/Press-Releases-Statements/Press-Release-View/Article/3608324/us-and-international-partners-issue-recommendations-to-secure-software-products/

[12] "IEC 62443," international Electrotechnical Commission (IEC) Standards.

[13] International Electrotechnical Commission, "Understanding IEC 62443," https://www.iec.ch/blog/understanding-iec-62443, accessed: July 16, 2023.

[14] S. Hu, B. Hua, and Y. Wang, "Comprehensiveness, automation and lifecycle: A new perspective for rust security," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, 2022, pp. 982–991.

[15] Z. Hanley, "Rust Won't Save Us: An Analysis of 2023's Known Exploited Vulnerabilities," Feb. 2024. [Online]. Available: https://www.horizon3.ai/attack-research/red-team/analysis-of-2023s-known-exploited-vulnerabilities/

[16] Glitchbyte, "Rust wont save us, but its ideas will," Feb. 2024. [Online]. Available: https://glitchbyte.io/posts/rust-wont-save-us/

[17] J. Getreu, "Embedded System Security with Rust - Case Study of Heartbleed."

[18] "How Rust Prevents Out of Bound Reads/Writes." [Online]. Available: https://ancat.github.io//rust/2017/01/21/rust-out-of-bounds.html

[19] SANS Institute, "Top 25 Software Errors," https://www.sans.org/top25-software-errors/, accessed: July 16, 2023.

[20] OWASP Foundation, "OWASP Top Ten," https://owasp.org/www-project-top-ten/, accessed: July 16, 2023.

[21] M. Howard, D. LeBlanc, and J. Viega, *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. New York: McGraw-Hill, 2005, accessed: July 16, 2023.

[22] T. Espinha Gasiba, U. Lechner, M. Pinto-Albuquerque, and D. Méndez, "Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2021, pp. 241–252, accessed: July 16, 2023.

[23] J. Sible and D. Svoboda, "Rust Software Security: A Current State Assessment," Carnegie Mellon University, Software Engineering Institute's Insights (blog), Dec 2022, accessed: July 16, 2023. [Online]. Available: https://doi.org/10.58012/0px4-9n81

[24] G. Wassermann and D. Svoboda, "Rust Vulnerability Analysis and Maturity Challenges," Carnegie Mellon University, Software Engineering Institute's Insights (blog), Jan 2023, accessed: July 16, 2023. [Online]. Available: https://doi.org/10.58012/t0m3-vb66

[25] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 763–779, accessed: July 16, 2023. [Online]. Available: https://doi.org/10.1145/3385412.3386036

[26] X. Zheng, Z. Wan, Y. Zhang, R. Chang, and D. Lo, "A Closer Look at the Security Risks in the Rust Ecosystem," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 34:1–34:30, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3624738

[27] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 156–161. [Online]. Available: https://doi.org/10.1145/3102980.3103006

[28] I. J. S. 22, "ISO/IEC TR 24772-1:2019 - Programming languages — Guidance to avoiding vulnerabilities in programming languages — Part 1: Language-independent guidance," Online, 12 2019, accessed: July 16, 2023. [Online]. Available: https://www.iso.org/standard/71091.html

[29] T. M. Corporation, "Common Weakness Enumeration (CWE)," Online, 2023, accessed: July 16, 2023. [Online]. Available: https://cwe.mitre.org/

[30] ANSSI, "Publication: Programming Rules to Develop Secure Applications With Rust," https://www.ssi.gouv.fr/guide/programming-rules-to-develop-secure-applications-with-rust/, 2023, (accessed July 16, 2023).

[31] "Rust - Analysis Tools," 2023, accessed: July 16, 2023. [Online]. Available: https://analysis-tools.dev/tag/rust

[32] R. Bagnara, A. Bagnara, and P. M. Hill, "Coding Guidelines and Undecidability," *arXiv*, Dec 2022, accessed: July 16, 2023. [Online]. Available: http://arxiv.org/abs/2212.13933

[33] T. Espinha Gasiba, U. Lechner, M. Pinto-Albuquerque, and D. Mendez Fernandez, "Awareness of Secure Coding Guidelines in the Industry - A First Data Analysis," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2020, pp. 345–352, accessed: July 16, 2023.

[34] T. Espinha Gasiba, U. Lechner, and M. Pinto-Albuquerque, "Sifu - a Cybersecurity Awareness Platform with Challenge Assessment and Intelligent Coach," *Cybersecurity*, vol. 3, no. 1, p. 24, 12 2020, accessed: July 16, 2023.

[35] S. Amburi, "Sathwik-Amburi/secure-software-development-with-rust: Secure Software Development with Rust," https://github.com/Sathwik-Amburi/secure-software-development-with-rust, Aug. 2023, last accessed: 2023-08-14. [Online]. Available: https://doi.org/10.5281/zenodo.8247155

[36] M. Noseda, F. Frei, A. Rüst, and S. Künzli, "Rust for secure iot applications: why c is getting rusty," in *Embedded World Conference*. Nuremberg, Germany: WEKA, June 2022. [Online]. Available: https://digitalcollection.zhaw.ch/handle/11475/25046

[37] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, sep 2021. [Online]. Available: https://doi.org/10.1145/3466642

[38] "National vulnerability database: Cve-2021-29922," https://nvd.nist.gov/vuln/detail/cve-2021-29922, May 2021, accessed: 2024-06-10.

[39] "Rust Command Injection: Examples and Prevention." [Online]. Available: https://www.stackhawk.com/blog/rust-command-injection-examples-and-prevention/

[40] "Rust CSRF Protection Guide: Examples and How to Enable It." [Online]. Available: https://www.stackhawk.com/blog/rust-csrf-protection-guide-examples-and-how-to-enable-it/

[41] SergioBenitez, "Add (more) csrf protection," https://github.com/rwf2/Rocket/issues/14, 2016, accessed: 24th Feb.

[42] "Rust XSS Guide: Examples and Prevention." [Online]. Available: https://www.stackhawk.com/blog/rust-xss-guide-examples-and-prevention/

[43] "Cross-site scripting (xss) affecting ammonia package." [Online]. Available: https://security.snyk.io/vuln/SNYK-RUST-AMMONIA-2929007

[44] OWASP Foundation, "Cross Site Scripting (XSS) Prevention Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html, Access Year.

[45] D. K. Sohr, "Architectural Aspects of Software Security." [Online]. Available: https://user.informatik.uni-bremen.de/sohr/papers/HabilSyn.pdf

[46] L. Futcher and R. von Solms, "Guidelines for secure software development," *ACM*, p. 56–65, 2008. [Online]. Available: https://doi.org/10.1145/1456659.1456667

[47] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Communications of the ACM*, vol. 17, no. 7, 07 1974, presented at the Fourth ACM Symposium on Operating System Principles (October 1973). [Online]. Available: https://www.cs.virginia.edu/~evans/cs551/saltzer/

[48] R. A. Khan, S. U. Khan, H. U. Khan, and M. Ilyas, "Systematic literature review on security risks and its practices in secure software development," *IEEE Access*, vol. 10, pp. 5456–5481, 2022.

[49] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428204

[50] K. Martin, I.-Y. Bang, J.-S. You, J.-W. Seo, and Y.-H. Paek, "A Study on Security Issues Due to Foreign Function Interface in Rust," *Proceedings of the Korea Information Processing Society Conference*, pp. 151–154, 2021, publisher: Korea Information Processing Society. [Online]. Available: https://koreascience.kr/article/CFKO202125036382349.page

[51] A. N. Evans, B. Campbell, and M. L. Soffa, "Is rust used safely by software developers?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 246–257. [Online]. Available: https://doi.org/10.1145/3377811.3380413

[52] H. Nina, J. A. Pow-Sang, and M. Villavicencio, "Systematic mapping of the literature on secure software development," *IEEE Access*, vol. 9, pp. 36 852–36 867, 2021.

[53] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, sep 2021, accessed: July 16, 2023. [Online]. Available: https://doi.org/10.1145/3466642

[54] The Rust Security Response WG, "Security advisory for rustc (CVE-2021-42574)," November 2021, accessed: July 16, 2023. [Online]. Available: https://blog.rust-lang.org/2022/01/20/cve-2022-21658.html

[55] ——, "Security advisory for the standard library (CVE-2022-21658)," January 2022, accessed: July 16, 2023. [Online]. Available: https://blog.rust-lang.org/2022/01/20/cve-2022-21658.html

[56] ——, "Security advisory for the regex crate (CVE-2022-24713)," March 2022, accessed: July 16, 2023. [Online]. Available: https://blog.rust-lang.org/2022/03/08/cve-2022-24713.html

[57] ——, "Security advisories for Cargo (CVE-2022-36113, CVE-2022-36114)," September 2022, accessed: July 16, 2023. [Online]. Available: https://blog.rust-lang.org/2022/09/14/cargo-cves.html

[58] ——, "Security advisory for Cargo (CVE-2022-46176)," January 2023, accessed: July 16, 2023. [Online]. Available: https://blog.rust-lang.org/2023/01/10/cve-2022-46176.html

[59] ——, "Security advisory for Cargo (CVE-2023-38497)," August 2023, accessed: June 10, 2024. [Online]. Available: https://blog.rust-lang.org/2023/08/03/cve-2023-38497.html

[60] ——, "Security advisory for Cargo (CVE-2024-24576)," April 2024, accessed: June 10, 2024. [Online]. Available: https://blog.rust-lang.org/2024/04/09/cve-2024-24576.html

[61] The Rust Project Developers, "rust-clippy: A bunch of lints to catch common mistakes and improve your rust code," https://github.com/rust-lang/rust-clippy, 2024, rust language. [Online]. Available: https://github.com/rust-lang/rust-clippy

[62] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 2183–2196. [Online]. Available: https://doi.org/10.1145/3460120.3484541

[63] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, Aug. 2021, pp. 597–616. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/fulton

[64] SonarSource, "SonarQube," https://www.sonarqube.org, [retrieved July 16, 2023]. [Online]. Available: https://www.sonarqube.org

[65] P. C. van Oorschot, "Memory errors and memory safety: A look at java and rust," *IEEE Security & Privacy*, vol. 21, no. 3, pp. 62–68, 2023.

[66] J. Jacoby and M. S. Matell, "Three-Point Likert Scales Are Good Enough," *Journal of Marketing Research*, vol. 8, no. 4, pp. 495–500, 1971, accessed: July 16, 2023. [Online]. Available: https://doi.org/10.1177/002224377100800414