I Think This is the Beginning of a Beautiful Friendship - On the Rust Programming Language and Secure Software Development in the Industry

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

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

***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 also compare Rust to C, C++, and Java, three widely used programming languages in the industry, regarding potential software vulnerabilities. Our results are based on a literature review, interviews with industrial cybersecurity experts, and an analysis of existing static code analysis tools. 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.**

Keywords–Cybersecurity*; *Software development*; *Industry*; *Software*; *Vulnerabilities.

I. INTRODUCTION

Rust, a systems programming language that originated in 2010, has significantly increased in popularity over the past decade. According to a market overview survey by Yalantis [1], 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, according to Garcia [2]. Stack Overflow [3] notes that developers appreciate Rust's focus on system-level details since it helps prevent null and dangling pointers and its memory safety without needing a garbage collector. 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 [4], 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. Rust's adoption in Linux Kernel development [5], [6] underlines its growing significance in an industrial context. Major platforms like Google have started including Rust in systems, such as Android [7], and forums like RustSec [8] provide real-time updates and insights into the current state of Rust security.

Rust promotes itself as being safer than traditional languages like 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 [9], [10], the notion of safety in Rust must be understood not just from a memory management perspective but also from a security standpoint.

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, 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 SysAdmin, Audit, Network, Security (SANS) Institute TOP 25 [11], Open Web Application Security Project (OWASP) 10 [12], and 19 Deadly Sins [13], and
2) Analyzing past known vulnerabilities in the Rust language and its ecosystem.

This study's contributions are as follows: firstly, through the present work, the authors aim to raise awareness, as defined by Gasiba et al. [14], about Rust security and its

 ISBN: 978-1-68558-113-8

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. [15]. 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.

In 2023, Wassermann et al. [16] 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. [17] 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.

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 [18] standard, "Secure Coding Guidelines Language Independent," provides guidelines suitable for multiple programming languages, including Rust. ISO/IEC 62.443 [9], especially sections 4-1 and 4-2, sets the industry standard for secure software development [10]. The Common Weakness Enumeration (CWE) by MITRE [19] 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" [20], 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 [7]. 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 [21]. These tools play a crucial role in the secure software development lifecycle.

Community-driven initiatives like RustSec [8] offer advisories on vulnerabilities in Rust crates. 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. [22] 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. [23] 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 [24] was developed in line with these challenges. Sifu 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.

 ISBN: 978-1-68558-113-8

III. METHODOLOGY

Our research methodology, aimed at examining the security in the Rust programming language compared to Java and C, and its interaction with security assessment tools, was composed of four main stages:

- Literature Exploration
- Interviews with Security Experts
- Mapping to CWE/SANS, OWASP, and 19 Deadly Sins
- Analysis with Rust/SAST Tools

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, 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 2023.

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 interviews 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 were conducted 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. 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 [25] 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 (2023)

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

 ISBN: 978-1-68558-113-8

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, 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.

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 of how the Rust language can offer protection against these vulnerabilities, according to the OWASP standard from 2021:

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%

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%

- **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.
- **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 ap-

proach to handling strings can help prevent injection attacks. However, poor programming practices may still result in these attacks; see PoC code in [25].

- **A04-Insecure Design (UP):** This vulnerability is more about the design of the application rather than the language itself. While Rust offers memory safety, 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.

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. 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:

- **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 [25].
- **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 [25].
- **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, handled by web frameworks rather than the language itself.

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%

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.

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.

D. 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 fixes them once discovered [8], [26]. However, Rust’s security advisory only addresses six of these vulnerabilities: CVE-2021-42574 [27], CVE-2022-21658 [28], CVE-2022-24713 [29], CVE-2022-36113 [30], CVE-2022-36114 [30], and CVE-2022-46176 [31].

The most recent CVE acknowledged by the Rust security advisory on their blog is CVE-2022-46176 [31]. 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.

E. 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 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.

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

 ISBN: 978-1-68558-113-8

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 [32], 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. 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.

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) [33], who argued that "Three-point Likert scales are good enough."

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 of-

 ISBN: 978-1-68558-113-8

fers 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. In further research, the authors would like to understand the security consequences of Rust's high learning curve through comparative studies of software projects developed in different programming languages.

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

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

[3] 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/

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

[5] 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. [Online]. Available: https://sdtimes.com/software-development/rusts-addition-to-the-linux-kernel-seen-as-enormous-vote-of-confidence-in-the-language/

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

[7] 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

[8] “RustSec Advisory Database,” 2023, accessed: July 16, 2023. [Online]. Available: https://rustsec.org/advisories/

[9] “IEC 62443,” international Electrotechnical Commission (IEC) Standards.

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

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

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

[13] 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.
Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), 2021, pp. 241–252, accessed: July 16, 2023.

[14] 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*

[15] 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

[16] 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

[17] 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

[18] 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

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

[20] 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).

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

[22] 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

[23] 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.

[24] 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.

[25] 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

[26] 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

[27] 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

[28] ——, “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

[29] ——, “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

[30] ——, “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

[31] ——, “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

[32] SonarSource, “SonarQube,” https://www.sonarqube.org, [retrieved July 16, 2023]. [Online]. Available: https://www.sonarqube.org

[33] 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

 ISBN: 978-1-68558-113-8