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
Siemens AG
Munich, Germany
tiago.gasiba@siemens.com

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

Project Member 1
Rabib Jahin Ibn Momin (0424052024)

Project Member 2
Md Zarzees Uddin Shah Chowdhury (0424052042)

Abstract

This study explores the security implications of adopting the Rust programming language in industrial settings. While Rust is widely acclaimed for its memory safety and concurrency features, its security aspects have not been thoroughly investigated. Through a combination of literature review, expert interviews, and vulnerability assessments aligned with industry security standards, this study analyzes how Rust addresses common software vulnerabilities in comparison to traditional languages like C, C++, and Java. The findings reveal that while Rust offers significant improvements in security, particularly in preventing memory-related vulnerabilities and data races, it is not entirely free from risks. To further investigate Rust's security claims, we conducted an analysis of the paper, implemented various code experiments to observe Rust's behavior, and evaluated its real-world effectiveness. We also explored Rust as a backend programming language, highlighting its performance, safety guarantees, and suitability for building secure and scalable systems. As a language without garbage collection, Rust aims to provide C-like performance while ensuring greater security and memory safety. This study bridges the gap between Rust's theoretical safety features and its practical application in real-world industrial environments.

Keywords: Rust Programming Language, Secure Software Development, Memory Safety, Vulnerability Analysis, Static Application Security Testing (SAST), Industrial Cybersecurity, CWE/SANS/OWASP Mapping, Unsafe Code Mitigation, Concurrency Safety, Secure Coding Practices

1 Introduction

The Rust programming language, emerging in 2010, has rapidly gained prominence in industrial software development due to its emphasis on *memory safety* and *concurrency*. With adoption in critical systems such as the Linux Kernel and Android, Rust positions itself as a robust alternative to traditional languages like C and C++. Its compile-time guarantees against memory corruption, data races, and null pointer dereferences address long-standing vulnerabilities pervasive in systems programming. However, while Rust's safety mechanisms are widely celebrated, a systematic evaluation of its security posture in industrial contexts—particularly in critical infrastructure adhering to standards like *IEC 62443* remains underexplored.

This paper investigates Rust's efficacy in mitigating software vulnerabilities compared to C, C++, and Java, languages entrenched in industrial practice. Despite Rust's advancements, conflating *safety* with *security* risks overlooking non-memory-related flaws, such as logic errors, insecure configurations, and web-layer vulnerabilities (e.g., Cross-Site Scripting). Their research addresses this gap through a multi-faceted methodology:

- **Literature Review**: Analyzing academic and gray literature (2010–2023) on Rust's security model, vulnerabilities, and tooling.
- **Expert Interviews**: Gathering insights from industry professionals with over a decade of experience in secure software development.
- **Vulnerability Mapping**: Aligning Rust's protections with frameworks like SANS Top 25, OWASP Top 10, and the *19 Deadly Sins of Software Security*.
- **Tool Analysis**: Evaluating Static Application Security Testing (SAST) tools for Rust against mature ecosystems like Java and C++.

Findings of this paper reveal that while Rust eliminates 24% of SANS Top 25 vulnerabilities (e.g., buffer overflows, race conditions) through language design, 48% remain unaddressed (e.g., path traversal, insecure authentication). Notably, vulnerabilities often stem from misuse of `unsafe` code blocks or design-level oversights unrelated to memory safety. Compared to C/C++ (no inherent protections) and Java (garbage-collected memory safety), Rust strikes a balance between low-level control and safety but necessitates complementary secure coding practices.

CYBER 2023 : The Eighth International Conference on Cyber-Technologies and Cyber-Systems, 2023

This work contributes to academia by dissecting Rust's security capabilities and limitations, while offering practitioners actionable insights for adopting Rust in compliance-driven environments. As industries increasingly prioritize cybersecurity resilience, understanding Rust's role—as both a safeguard and a potential risk—is critical for shaping future secure development paradigms.

2 Related Work

Research on Rust's security has evolved alongside its industrial adoption. Sible et al. [19] conducted a foundational analysis of Rust's memory and concurrency safety, highlighting limitations such as memory leaks and the need for holistic security practices. Wassermann et al. [21] expanded this by emphasizing vulnerabilities arising from misaligned design assumptions and advocating for ecosystem maturity. Their work stresses the importance of analyzing real-world vulnerabilities, even without source code access, to mitigate risks in Rust-based systems.

Qin et al. [15] empirically studied unsafe code usage in Rust projects, revealing that while developers minimize unsafe blocks, memory-safety bugs often stem from interactions between safe and unsafe code. Their findings underscore the complexity of Rust's lifetime system and the risks of integrating unsafe code for performance. Xu et al. [22] further analyzed all Rust common vulnerability and exposures (CVEs), concluding that memory-safety challenges persist despite Rust's compile-time guarantees, particularly when developers bypass safety mechanisms.

Security Standards and Industrial Guidelines. Industrial adoption of Rust necessitates alignment with standards like *IEC 62443* [10] for critical infrastructure and *ISO/IEC TR 24772* [11] for secure coding. The *Common Weakness Enumeration (CWE)* [13] framework provides a unified taxonomy of software weaknesses, while ANSSI's guidelines for Rust [4] offer tailored rules to address its safety-security interplay.

Tools and Ecosystem Maturity. The Rust ecosystem includes tools like `RustSec` [2] for tracking crate vulnerabilities and SAST tools listed in `Analysis Tools` [1]. Google's integration of Rust into Android [8] exemplifies its industrial use to mitigate memory-safety bugs. However, maturity gaps persist compared to Java/C++ ecosystems, which leverage tools like `SonarQube` [20] for compliance.

Secure Coding Practices and Awareness. Gasiba et al. [7] identified a gap between developers' intent to follow secure coding guidelines and their practical knowledge. Their work inspired platforms like *Sifu* [6], which gamifies cybersecurity education. Bagnara et al. [5] explored the undecidability of secure coding rules (e.g., input validation), emphasizing that no language can fully automate vulnerability prevention. Jacoby et al. [12] validated the three-point vulnerability classification (RD/SG/UP) by demonstrating the sufficiency of Likert-style scales for such analyses.

Rust CVEs and Mitigations. The Rust Security Response WG addresses vulnerabilities like `CVE-2021-42574` [16] (Unicode trojans) and `CVE-2022-46176` [17] (Cargo SSH vulnerabilities). Amburi's PoC code [3] demonstrates exploitable weaknesses (e.g., SQL injection) even in Rust, reinforcing the need for complementary safeguards.

Collectively, these works contextualize Rust's strengths but leave gaps in systematic vulnerability analysis across industrial standards. Their study bridges this by mapping Rust's protections to SANS Top 25, OWASP Top 10, and the *19 Deadly Sins* [9], while addressing tooling and awareness challenges specific to industrial adoption.

3 Methodology

The study evaluates Rust's security capabilities through a multi-phase approach, combining qualitative and quantitative analyses. The methodology addresses two research questions: **RQ1:** *To what extent does Rust mitigate vulnerabilities compared to C, C++, and Java?* **RQ2:** *How effective are existing tools and practices for secure Rust development in industrial contexts?*

3.1 Research Design

The methodology comprises four stages:

- **Literature Review**: They analyzed academic and gray literature (2010–2023) from ACM, IEEE Xplore, and Google Scholar using keywords like `"Rust Security"` and `"SAST Tools"`. This included foundational works by Qin et al. [15] on unsafe code and Gasiba et al. [7] on secure coding awareness.
- **Expert Interviews**: Semi-structured interviews were conducted with five industry security experts (10+ years of experience) and two open-source Rust contributors. Discussions focused on Rust's adoption challenges, vulnerability patterns, and tooling gaps.
- **Vulnerability Mapping**: They mapped Rust's protections to three frameworks:
  1. *SANS Top 25* [18] and *CWE* [13] for system-level weaknesses.
  2. *OWASP Top 10* [14] for application-layer vulnerabilities.
  3. *19 Deadly Sins of Software Security* [9] for design flaws.
- **Static Analysis**: They evaluated Rust's SAST tools (e.g., `Clippy`, `RustSec`) against Java (`SonarQube` [20]) and C++ (`cppcheck`) ecosystems.

3.2 Vulnerability Classification

They categorized vulnerabilities using a three-point scale inspired by Jacoby et al. [12]:

1. **Rare and Difficult (RD)**: Fully mitigated by Rust's design (e.g., buffer overflows [22]) unless `unsafe` blocks are used.
2. **Safeguarded (SG)**: Partially addressed by Rust or dependent on libraries (e.g., SQL injection via `rusqlite` [3]).
3. **Unprotected (UP)**: Require external mitigations (e.g., insecure authentication [4]).

3.3 Validation

The authors developed Proof-of-Concept (PoC) code [3] demonstrating vulnerabilities in Rust (e.g., command injection, TOCTOU). These were analyzed using SAST tools to assess detection efficacy. Industrial experts validated findings against IEC 62443 [10] compliance requirements.

4 Results

The analysis reveals Rust's strengths and limitations in mitigating vulnerabilities across industrial standards. Below, we summarize key findings from vulnerability mappings, CVEs, and SAST tool comparisons.

4.1 SANS Top 25 Vulnerabilities in Rust

Table 1 categorizes Rust's protection levels for SANS Top 25 CWEs. Rust fully mitigates 24% of vulnerabilities (e.g., buffer overflows [22]) through compile-time checks, while 48% remain unprotected (e.g., path traversal).

4.2 Comparison with C, C++, and Java

Table 2 highlights Rust's advantages over C/C++ (no inherent protections) and Java (partial safeguards via garbage collection). For example, Rust mitigates all memory-safety CWEs (e.g., CWE-119) unless `unsafe` is used.

4.3 Explanation with Code

We will understand the classification via 2 important examples from the table: CWE-787 & CWE-362

4.3.1 CWE-787: Out-of-Bounds Write Vulnerability

Classification. Rust's ownership and borrowing system prevents out-of-bounds writes at compile time, unless `unsafe` blocks are used.

```
fn main() {
    let mut arr = [1, 2, 3, 4, 5];

    // Attempt to write out of bounds (will cause a compile-time error)
    arr[5] = 6; // Index 5 is out of bounds for an array of length 5
}
```

Explanation:

- Rust's compiler enforces bounds checking, preventing out-of-bounds writes.
- If `unsafe` blocks are used, the responsibility shifts to the programmer, but this is discouraged unless absolutely necessary.

Table 1. 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	CSRF			•
CWE-476	NULL Pointer Dereference	•		
CWE-190	Integer Overflow		•	
CWE-798	Improper Authentication			•
CWE-259	Use of Hard-coded Credentials			•
CWE-362	Missing Authorization			•
CWE-77	Command Injection		•	
CWE-119	Buffer Overflow	•		
CWE-276	Incorrect Default Permissions			•
CWE-918	Server-Side Request Forgery			•
CWE-362	Race Condition	•		
CWE-611	Improper Restriction of XXE			•
CWE-94	Code Injection		•	

Protection Levels: Rare and Difficult (RD) 24%, Safeguarded (SG) 28%, Unprotected (UP) 48%.

C does not provide built-in memory safety mechanisms, making out-of-bounds writes a common vulnerability.

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    // Out-of-bounds write (no compile-time or runtime checks)
    arr[5] = 6; // Undefined behavior, may corrupt memory

    printf("%d\n", arr[5]); // May print garbage or crash
    return 0;
}
```

Explanation:

- C does not enforce bounds checking, leading to undefined behavior if an out-of-bounds write occurs.
- This makes C programs highly vulnerable to memory corruption attacks.

Table 2. SANS Top 25 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-352			•			•			•
CWE-434			•			•			•
CWE-476			•		•			•	
CWE-502			•			•			•
CWE-190			•			•	•		
CWE-798			•			•			•
CWE-287			•			•			•
CWE-259			•			•			•
CWE-862			•			•			•
CWE-77			•			•			•
CWE-306			•			•			•
CWE-119			•		•		•		
CWE-276			•			•			•
CWE-918			•			•			•
CWE-362			•			•		•	
CWE-400			•		•				•
CWE-611			•			•			•
CWE-94			•			•			•

Protection Levels:

C: RD 0%, SG 0%, UP 100%; C++: RD 0%, SG 24%, UP 76%; Java: RD 20%, SG 28%, UP 52%.

C++ provides some safeguards (e.g., `std::vector` with bounds-checked access), but raw arrays and pointers are still unsafe.

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> arr = {1, 2, 3, 4, 5};

    // Safe access with bounds checking (throws an exception if out of bounds)
    try {
        arr.at(5) = 6; // Throws std::out_of_range exception
    } catch (const std::out_of_range& e) {
        std::cerr << "Out of bounds access: " << e.what() << std::endl;
    }

    // Unsafe access with raw arrays (no bounds checking)
    int raw_arr[5] = {1, 2, 3, 4, 5};
    raw_arr[5] = 6; // Undefined behavior, may corrupt memory

    return 0;
}
```

Explanation:

- C++ provides safeguards like `std::vector::at()`, which performs bounds checking and throws exceptions for out-of-bounds access.
- However, raw arrays and pointers in C++ are still unsafe and require careful handling to avoid vulnerabilities.

4.3.2 CWE-362: Race Condition Vulnerability Classification.

We classify the **Race Condition** vulnerability in Rust, C, C++, and Java based on their concurrency safety mechanisms: Rust's ownership and borrowing system, combined with its concurrency primitives, makes race conditions rare and difficult to occur.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Result: {}", *counter.lock().unwrap());
}
```

Explanation:

- Rust enforces strict concurrency rules at compile time, preventing data races.
- The `Mutex` and `Arc` types ensure safe shared access to data across threads.

C++ provides concurrency primitives (e.g., `std::mutex`), but race conditions are still common if not used correctly.

```cpp
#include <iostream>
#include <thread>
#include <vector>

int counter = 0;
```

```
void increment() {
    for (int i = 0; i < 1000; i++) {
        counter++; // Race condition: unprotected access
    }
}

int main() {
    std::vector<std::thread> threads;

    for (int i = 0; i < 10; i++) {
        threads.emplace_back(increment);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Result: " << counter << std::endl; // Likely incorrect due to race conditions
    return 0;
}
```

Explanation:

- C++ provides tools like `std::mutex`, but it is up to the programmer to use them correctly.
- Without proper synchronization, race conditions are common in C++ programs.

Java provides built-in concurrency safeguards (e.g., `synchronized` blocks, `java.util.concurrent` utilities), but race conditions can still occur if not used properly.

```
public class Main {
    private static int counter = 0;

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[10];

        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    synchronized (Main.class) { // Safeguard: synchronized block
                        counter++;
                    }
                }
            });
            threads[i].start();
        }

        for (Thread t : threads) {
            t.join();
        }

        System.out.println("Result: " + counter); // Correct due to synchronization
    }
}
```

Explanation:

- Java provides built-in mechanisms like `synchronized` blocks and the `java.util.concurrent` package to prevent race conditions.
- However, improper use of these mechanisms can still lead to race conditions.

4.4 OWASP Top 10 Mapping

As shown in Table 3, Rust partially safeguards 50% of OWASP vulnerabilities (e.g., injection via libraries) but offers no protection against design-level flaws (A04, A05).

Table 3. OWASP Top 10 Mapping 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			•

Protection Levels: RD 0%, SG 50%, UP 50%.

4.5 19 Deadly Sins of Software Security

Rust fully mitigates 21% of the "sins" (e.g., buffer overflows) and partially addresses 47% (e.g., SQL injection). Unprotected issues (32%) include insecure configurations and SSRF (Table 4).

4.6 CVEs and SAST Tool Effectiveness

Rust's security advisories address critical CVEs like `CVE-2022-46176` [17] (Cargo SSH flaw). However, SAST tools for Rust (e.g., Clippy) lag behind Java's SonarQube [20] in detecting logic flaws. Only 6 of 400+ Rust CVEs are actively tracked by `RustSec` [2].

5 Discussion

Our study underscores Rust's transformative potential in secure software development while exposing critical gaps that challenge its industrial adoption. Below, we contextualize our results, compare them with prior work, and outline implications for academia and industry.

5.1 Rust's Security Trade-offs

Rust's compile-time guarantees eliminate entire classes of vulnerabilities (e.g., buffer overflows, data races) that plague C/C++ systems [15]. However, our analysis reveals that 48% of SANS Top 25 vulnerabilities remain unaddressed (Table 1),

Table 4. Security Flaws vs. 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)			•

Protection Levels: RD: 21%, SG: 47%, UP: 32%.

echoing Wassermann et al. [21], who caution against conflating memory safety with holistic security. For instance, `unsafe` code, used in 34% of Rust crates [22], reintroduces risks akin to C, particularly when developers misapply lifetimes or bypass borrow-checker constraints.

5.2 Comparative Analysis with Established Languages

While Rust outperforms C/C++ in memory safety, Java's garbage collection provides comparable protection against memory leaks (Table 2). However, Java's reliance on runtime checks incurs performance costs, whereas Rust's zero-cost abstractions align with industrial demands for efficiency [8]. This duality positions Rust as a viable candidate for systems requiring both safety and performance, such as embedded devices or critical infrastructure adhering to IEC 62443 [10].

5.3 Tooling and Ecosystem Challenges

The immaturity of Rust's SAST ecosystem, compared to Java's SonarQube [20] or C++'s Clang Analyzer, exacerbates vulnerability detection gaps. For example, only 6 of 400+ Rust CVEs are tracked in RustSec [2], leaving developers reliant on community-driven efforts. This aligns with Gasiba et al. [7], who identified tooling as a critical enabler of secure coding practices. Until Rust's tooling matures, organizations must supplement SAST with manual audits and adherence to guidelines like ANSSI's [4].

5.4 Practical Implications

Industries adopting Rust must prioritize:

- **Training**: Addressing Rust's steep learning curve through targeted programs, as misused lifetimes and ownership rules remain prevalent sources of bugs [15].
- **Secure Design**: Complementing Rust's safety with frameworks like OWASP Top 10 to mitigate web-layer flaws (Table 3).
- **Hybrid Approaches**: Using Rust for safety-critical components while leveraging Java/Python for higher-layer logic, as seen in Android [8].

5.5 Limitations and Future Work

Our three-point vulnerability scale (RD/SG/UP), validated by Jacoby et al. [12], simplifies complex interactions but may overlook context-specific risks. Future studies should:

- Investigate Rust-specific vulnerabilities (e.g., trait object misuse).
- Develop SAST tools for IEC 62443 compliance.
- Conduct longitudinal studies on Rust's security in large-scale industrial projects.

6 Critical Analysis

6.1 Strengths of the Research

Timeliness and Relevance: The research is highly relevant given Rust's increasing adoption in the software industry, particularly in security-critical areas such as the Linux Kernel and Android development. The study contributes to an area with limited systematic security analysis.

Comprehensive Approach: The methodology incorporates:

- Literature review
- Interviews with industry security experts
- Security standard mapping (CWE, SANS, OWASP)
- Static analysis tools

This multifaceted approach strengthens the validity of the findings.

Comparative Analysis: The study effectively compares Rust with C, C++, and Java, providing structured vulnerability classification using the categories: Rare and Difficult (RD), Safeguarded (SG), and Unprotected (UP). This categorization aids in understanding Rust's security posture.

Discussion of Real-World Vulnerabilities: By analyzing past Common Vulnerabilities and Exposures (CVEs) in Rust, the study underscores that while Rust mitigates memory safety issues, it remains vulnerable to other forms of software security risks.

Contribution to Industry and Academia: The paper raises awareness of Rust's security pitfalls and provides insights

for both researchers and industry practitioners, facilitating informed decision-making regarding Rust adoption.

6.2 Weaknesses and Limitations

Limited Empirical Evidence: The study lacks large-scale empirical testing. Conducting real-world security testing on Rust applications would enhance the robustness of the conclusions.

Small Sample Size in Interviews: Only five industry experts and two students were interviewed. A broader sample size would provide a more comprehensive perspective on Rust's security challenges.

Overemphasis on Memory Safety : While memory safety is a crucial aspect, the study could have further explored non-memory vulnerabilities, such as improper authentication and insecure cryptographic implementations.

Lack of Industry Case Studies: The absence of real-world case studies from companies using Rust for security reasons limits practical applicability. Including case studies would provide tangible insights into Rust's effectiveness.

Missing Discussion on Secure Development Lifecycle : The research does not address how Rust fits into a broader secure software development lifecycle (SDLC). Analyzing Rust's role in security testing, deployment, and CI/CD pipelines would be beneficial.

7 Our Experiment

7.1 Experimental Comparison: Rust vs other Backend Language (Node.js and Python Django)

To extend the authors' research on Rust's security in comparison to other programming languages, we introduce an additional analysis focusing on backend development. While the original study compares Rust against C, C++, and Java, our work expands this comparison by incorporating Node.js and Django, two widely used backend technologies. Our primary objective is to evaluate how well Rust protects against the SANS Top 25 Common Weakness Enumeration (CWE) security vulnerabilities compared to these backend alternatives. We analyzed the 6 among 25 vulnerabilities shown in the table 5 We categorize security protections using the same three-level classification system—Rare and Difficult (RD), Safeguarded (SG), and Unprotected (UP)—to maintain consistency with the original research. Through this extended study, we provide insights into Rust's security strengths and weaknesses within web and backend development, highlighting how it differs from frameworks and languages like Node.js (JavaScript-based) and Django (Python-based). This analysis is particularly relevant for industry professionals considering Rust for backend applications, helping them weigh security implications alongside performance and scalability factors.

Table 5. Vulnerability Classification for Rust, Node.js, and Django(Python Core Language)

CWE ID	Description	Rust	Node.js	Python
787	Out-of-bounds Write	RD	UP	RD
125	Out-of-bounds Read	RD	UP	RD
416	Use After Free	RD	UP	RD
190	Integer Overflow	RD	UP	RD
119	Buffer Overflow	RD	UP	RD
362	Race Condition	RD	UP	SG

7.2 Explanation with Code

Out-of-bounds Write (CWE-787)

- **Rust (RD):** Rust's ownership model and compile-time checks prevent out-of-bounds writes by ensuring all memory accesses are within bounds.

```
    let mut arr = [1, 2, 3];
// arr[5] = 10; // Compile-time error: index out of bounds
```

- **Node.js (UP):** JavaScript/Node.js does not inherently protect against out-of-bounds writes when interacting with native modules or external libraries.

```
    const buffer = Buffer.alloc(10);
buffer[15] = 255; // Out-of-bounds write (No immediate error, but unsafe)
```

- **Python (RD):** Python's dynamic memory management prevents direct memory manipulation, making out-of-bounds writes impossible in pure Python code.

```
    buffer = bytearray(b"ABC")
# buffer[5] = 65  # IndexError: index out of range
```

Use After Free (CWE-416)

- **Rust (RD):** Rust's ownership model ensures that memory is automatically deallocated when it goes out of scope, preventing use-after-free errors. The compiler enforces strict rules to ensure no references to freed memory exist.

```
let x = Box::new(42);
// drop(x); // Explicitly deallocates memory
// println!("{}", x); // Compile-time error: use of moved value
```

- **Node.js (UP):** JavaScript/Node.js relies on garbage collection to manage memory. However, native modules or external libraries can introduce use-after-free vulnerabilities if they improperly handle memory.

```
let buffer = Buffer.alloc(10);
buffer.fill(0);
buffer = null; // Memory is freed
// Accessing buffer here would lead to undefined behavior
```

- **Python (RD):** Python's garbage collector manages memory automatically, preventing use-after-free errors in pure Python code. Once an object is no longer referenced, it is deallocated safely.

```
class MyClass:
    def __init__(self):
        self.value = 42

obj = MyClass()
obj = None  # Garbage collector deallocates memory
# Accessing obj here would raise an error
```

Race Condition (CWE-362)

- **Rust (RD):** Rust's ownership model prevents data races at compile time by enforcing strict rules about mutable and shared references. This makes race conditions rare and difficult to occur in safe Rust code.

```
use std::sync::{Arc, Mutex};
use std::thread;

let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..10 {
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
}

for handle in handles {
    handle.join().unwrap();
}
// Safe from race conditions due to Mutex
```

- **Node.js (UP):** JavaScript/Node.js is single-threaded, which reduces the likelihood of race conditions. However, asynchronous operations can still lead to race conditions if shared state is not properly synchronized.

```
let counter = 0;

setTimeout(() => {
    counter++;
}, 100);

setTimeout(() => {
    console.log(counter); // May print 0 or 1 depending on timing
}, 100);
```

- **Python (SG):** Python provides synchronization primitives like locks (`threading.Lock`) to handle race conditions in multi-threaded programs. Developers must explicitly use these tools to safeguard against race conditions.

```
import threading

counter = 0
lock = threading.Lock()

def increment():
    global counter
    with lock:
        counter += 1

threads = []
for _ in range(10):
    thread = threading.Thread(target=increment)
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

print(counter)  # Always prints 10 (safe due to lock)
```

8 Conclusion and Future Work

This study explored the security implications of adopting Rust in industrial settings, highlighting both its strengths and limitations compared to languages like C, C++, Java, Node.js, and Python. The findings reinforced that Rust's ownership model, memory safety guarantees, and concurrency mechanisms significantly mitigate vulnerabilities such as buffer overflows, use-after-free errors, and race conditions. However, despite these advantages, Rust does not inherently protect against all security threats, particularly application-layer issues such as injection vulnerabilities and insecure authentication.

Our additional experiments extended the original study by evaluating Rust's security posture in backend development compared to Node.js and Django. The results indicated that while Rust provides strong safeguards against low-level

memory vulnerabilities, web frameworks in JavaScript and Python offer built-in protections against high-level security risks, such as cross-site scripting and command injection. This suggests that while Rust is well-suited for system-level security, additional precautions are necessary when using it in web development.

One of the key challenges identified is the relative immaturity of Rust's static analysis and security tooling compared to more established ecosystems like Java and C++. This gap makes it essential for developers to complement Rust's safety mechanisms with external security practices, such as secure design patterns, manual audits, and adherence to industry standards like IEC 62443 and OWASP Top 10.

For future work, several research directions remain open:

- Investigating Rust-specific security risks, including vulnerabilities that arise from misuse of unsafe code and trait object interactions.
- Enhancing Rust's security tooling by developing more robust static analysis tools capable of detecting design-level security flaws.
- Conducting large-scale empirical studies on Rust's security in industrial applications to assess its effectiveness over time.
- Exploring hybrid approaches that integrate Rust with other languages, leveraging its strengths in performance and security while mitigating its limitations in web-layer protections.

Overall, Rust continues to be a promising choice for secure software development, particularly in safety-critical and performance-sensitive applications. However, to fully harness its potential, a comprehensive security strategy incorporating both language-level guarantees and best practices from the broader cybersecurity domain is necessary.

References

[1] 2023. Rust - Analysis Tools. https://analysis-tools.dev/tag/rust Accessed: July 16, 2023.

[2] 2023. RustSec Advisory Database. https://rustsec.org/advisories/ Accessed: July 16, 2023.

[3] Sathwik Amburi. 2023. Secure Software Development with Rust. doi:10.5281/zenodo.8247155 Last accessed: August 14, 2023.

[4] ANSSI. 2023. Programming Rules to Develop Secure Applications With Rust. https://www.ssi.gouv.fr/en/guide/programming-rules-to-develop-secure-applications-with-rust/ Accessed: July 16, 2023.

[5] Roberto Bagnara, Alessandro Bagnara, and Patricia M. Hill. 2022. Coding Guidelines and Undecidability. *arXiv* (2022). arXiv:2212.13933

[6] Tiago Espinha Gasiba, Ulrike Lechner, and Maria Pinto-Albuquerque. 2020. Sifu - A Cybersecurity Awareness Platform with Challenge Assessment and Intelligent Coach. *Cybersecurity* 3, 1 (2020), 24. Accessed: July 16, 2023.

[7] Tiago Espinha Gasiba, Ulrike Lechner, Maria Pinto-Albuquerque, and Daniel Méndez. 2021. 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)*. 241–252.

[8] Google Security Team. 2022. Memory-Safe Languages in Android 13. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html Accessed: July 16, 2023.

[9] Michael Howard, David LeBlanc, and John Viega. 2005. *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill.

[10] International Electrotechnical Commission. 2023. *IEC 62443: Industrial Communication Networks - Network and System Security*. Technical Report. https://www.iec.ch/blog/understanding-iec-62443 Accessed: July 16, 2023.

[11] ISO/IEC. 2019. *ISO/IEC TR 24772-1:2019 - Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages*. Technical Report. https://www.iso.org/standard/71991.html Accessed: July 16, 2023.

[12] Jacob Jacoby and Michael S. Matell. 1971. Three-Point Likert Scales Are Good Enough. *Journal of Marketing Research* 8, 4 (1971), 495–500. doi:10.1177/002224377100800414

[13] MITRE Corporation. 2023. Common Weakness Enumeration (CWE). https://cwe.mitre.org/ Accessed: July 16, 2023.

[14] OWASP Foundation. 2021. OWASP Top Ten. https://owasp.org/www-project-top-ten/ Accessed: July 16, 2023.

[15] Bingyu Qin, Yan Chen, Zheng Yu, Limin Song, and Yuqing Zhang. 2020. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Languages Design and Implementation*. 763–779. doi:10.1145/3385412.3386036

[16] Rust Security Response Working Group. 2021. Security Advisory for Rustc (CVE-2021-42574). https://blog.rust-lang.org/2022/01/20/cve-2022-21658.html Accessed: July 16, 2023.

[17] Rust Security Response Working Group. 2023. Security Advisory for Cargo (CVE-2022-46176). https://blog.rust-lang.org/2023/01/10/cve-2022-46176.html Accessed: July 16, 2023.

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

[19] J. Sible and D. Svoboda. 2022. Rust Software Security: A Current State Assessment. https://doi.org/10.58012/0px4-9n81 Accessed: July 16, 2023.

[20] SonarSource. 2023. SonarQube. Software Tool. https://www.sonarqube.org Accessed: July 16, 2023.

[21] G. Wassermann and D. Svoboda. 2023. Rust Vulnerability Analysis and Maturity Challenges. https://doi.org/10.58012/t0m3-vb66 Accessed: July 16, 2023.

[22] Hui Xu, Zheng Chen, Mingliang Sun, Yangfan Zhou, and Michael R. Lyu. 2021. Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. *ACM Transactions on Software Engineering and Methodology* 31, 1 (2021). doi:10.1145/3466642