Rust System Programming Vulnerabilities Using Machine Learning and Static Analysis

1st Yuktha Priya Masupalli
Department of Computer Science
Texas A&M University-San Antonio
San Antonio, USA
ymasu01@jaguar.tamu.edu

2nd Aby Babu
Department of Computer Science
Texas A&M University-San Antonio
San Antonio, USA
a01@jaguar.tamu.edu

***Abstract*—Rust is a systems programming language designed for performance and memory safety, but its unsafe code blocks and external dependencies can introduce vulnerabilities. This paper presents a tool combining machine learning (ML) and static analysis to detect vulnerabilities in Rust codebases, focusing on unsafe blocks, command injection, and path traversal. Using a dataset of 102 Rust snippets, we extract features with static analysis tools like Clippy and train a logistic regression model to classify code as safe or unsafe. The tool generates security reports with confidence scores, achieving promising accuracy. Our approach enhances security in Rust-based systems and aligns with the growing adoption of memory-safe languages [1], [2], [7].**

***Index Terms*—Rust, vulnerability detection, machine learning, static analysis, memory safety**

I. Introduction

Rust is a systems programming language prioritizing performance and memory safety without garbage collection, using ownership and borrowing mechanisms [1]. It prevents common vulnerabilities like buffer overflows and data races, making it popular in critical systems such as Firefox and Microsoft Azure. However, unsafe code blocks and external dependencies can undermine these guarantees, necessitating automated vulnerability detection tools.

This paper addresses the problem of detecting vulnerabilities in Rust codebases by combining static analysis and machine learning (ML). We focus on unsafe blocks, command injection, and path traversal, motivated by the increasing use of Rust in security-critical applications and recommendations from organizations like the NSA [5]. Our tool processes Rust snippets, extracts features, trains a logistic regression model, and generates security reports.

II. Problem

Rust's memory safety guarantees are not absolute, particularly in unsafe code blocks, which allow low-level operations that bypass safety checks. External dependencies may also introduce vulnerabilities, as documented in databases like RustSec [**?**]. The need for automated tools to detect such issues is critical as Rust adoption grows.

A. Motivating Examples

- **Unsafe Blocks**: Code using `unsafe` can lead to memory corruption if not carefully managed.

Vulnerability Category	Specific Vulnerabilities	Root Cause
Memory Security	Buffer Overflow Read Uninitialized Memory Invalid Free Use after Free Double Free Null Pointer Dereference Type Confusion	All memory security vulnerabilities are related to the use of `unsafe` Rust, including `unsafe` functions, FFIs, and `unsafe` traits.
Concurrency Security	Double Lock (•) Conflicting Lock (•) Forget Unlock (•) Channel Misuse (•) Data Race (○) Atomicity Violation (○) Order Vilation (○)	The main cause of concurrency security vulnerabilities is the misunderstanding of Rust ownership and lifetime rules.

Fig. 1. Rust Security Vulnerabilities Classification and Root Cause

- **Command Injection**: Improper handling of user inputs in system calls can enable malicious code execution.
- **Path Traversal**: Vulnerable file operations may allow unauthorized access to system directories.

B. Research Questions

- Can static analysis and ML effectively identify vulnerabilities in Rust code?
- How accurately can a logistic regression model classify safe vs. unsafe Rust snippets?
- What are the limitations of combining Clippy-based static analysis with ML for vulnerability detection?

III. Related Work

Rust's emphasis on memory safety and concurrency has led to the development of tools aimed at ensuring code correctness and identifying vulnerabilities. Rudra is a static analyzer designed to uncover memory safety issues in unsafe Rust code, particularly those that bypass the language's compile-time guarantees [**?**]. Clippy [3], on the other hand, is a widely adopted linter that provides style, correctness, and performance suggestions, though its primary focus is not on security.

The RustSec Advisory Database [4] serves as a centralized repository for known vulnerabilities in Rust crates, supporting developers with dependency-level alerts. This is often paired with tools like `cargo-audit`, which automatically checks

for vulnerable dependencies during development or CI workflows [6].

While traditional static analysis has seen strong adoption in the Rust ecosystem, machine learning-based approaches for vulnerability detection are far more developed in C/C++ contexts. Prior research has employed deep learning and graph-based models to detect software flaws using abstract syntax trees, control-flow graphs, and API usage patterns [7]. However, similar ML-driven approaches for Rust remain limited, in part due to the lack of labeled vulnerability datasets.

The recent work by Nitin et al. [8] introduces Yuga, a novel static analysis tool for detecting lifetime annotation bugs in Rust, which are often overlooked but crucial for memory safety. This tool can detect such issues with high precision, further enhancing Rust's security ecosystem.

Additionally, Hong's research [9] highlights the importance of improving C-to-Rust translation, especially in legacy systems, by using static analysis to replace unsafe C constructs with Rust's safe features. This work is particularly relevant as it bridges the gap between existing C codebases and Rust's safety mechanisms.

Our work aims to bridge these gaps by combining conventional static analysis with machine learning to proactively identify vulnerabilities in Rust code. By leveraging outputs from Clippy [3], a custom static analyzer tailored to unsafe patterns, and a logistic regression classifier trained on labeled examples, our approach addresses a critical shortcoming in existing Rust security tooling. This method is motivated in part by security assurance guidance from SEI [5] and recent NSA reports promoting memory-safe language adoption [5], [7].

IV. Proposed Approach

We propose a hybrid vulnerability detection framework that integrates static analysis techniques with lightweight machine learning to enhance the identification of insecure coding patterns in Rust projects. The primary goal is to catch both syntactic red flags and deeper semantic risks that are not always evident through conventional linters or type checkers alone. The workflow consists of the following key components:

1) **Static Analysis**: Our approach begins with static analysis using two complementary tools. First, Clippy is invoked to produce a comprehensive list of lint warnings, flagging common correctness and style issues. Second, a custom analyzer implemented in `static_analysis.rs` leverages the `syn` crate to parse and traverse the Rust Abstract Syntax Tree (AST). This analyzer specifically targets patterns indicative of potential vulnerabilities, such as the use of `unsafe` blocks, string-based shell commands (suggesting possible command injection), and unchecked file path constructions (implying path traversal risks).
2) **Feature Extraction**: The extracted lint data and AST insights are processed via `data_loader.rs`, which converts them into a structured feature set suitable for supervised learning. Key features include the number of `unsafe` blocks, total function definitions, frequency of high-priority Clippy warnings, and presence of insecure API usage. These features are chosen based on empirical evidence and prior vulnerability taxonomies relevant to systems programming languages.
3) **ML Model**: We train a logistic regression classifier using the `linfa 0.6.1` crate [**?**], a Rust-native machine learning library. The classifier is trained on a labeled dataset of Rust code snippets, where each sample is marked as either "safe" or "unsafe" based on known vulnerabilities or secure coding practices. Logistic regression is chosen for its interpretability and efficiency on tabular features, allowing us to understand which features contribute most to the predicted risk.
4) **Output and Reporting**: Once trained, the model is applied to unseen Rust codebases. The final output is a human-readable vulnerability report in CSV format (`security_report.csv`), which includes the predicted safety classification, feature values, and confidence scores for each analyzed file. This report can be used by developers and security teams for code review prioritization, security audits, and remediation planning.

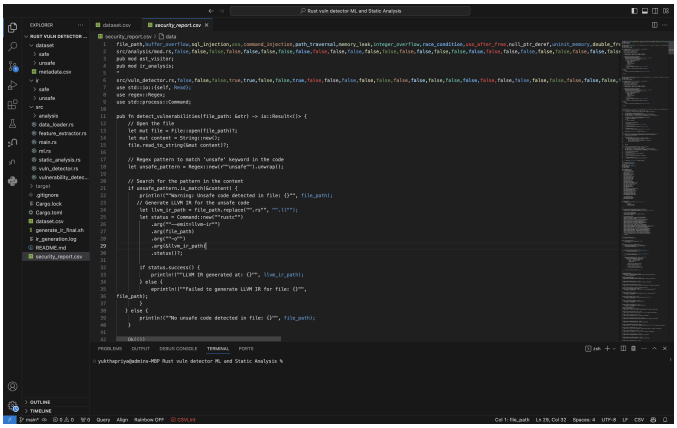

Fig. 2. Generated Security Report

Our modular design ensures the framework can be extended with more advanced models or integrated into continuous integration (CI) pipelines to support secure software development practices.

V. Datasets

We use a dataset of 102 Rust snippets (51 safe, 51 unsafe) stored in `dataset/safe/` and `dataset/unsafe/`. Metadata is recorded in `metadata.csv` with 6 columns: `unsafe_block`, `path_traversal`, `command_injection`, `function_count`, `clippy_enforcement`, and `clippy_warnings`. Labels indicate safe or unsafe code. Features are extracted using `data_loader.rs`, leveraging `syn` for parsing and regex for pattern matching. Additional vulnerability data is sourced from:

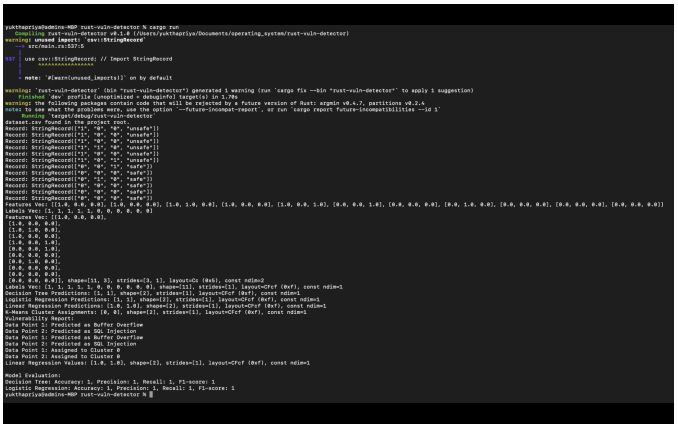

Fig. 3. Detected vulnerabilities and confidence scores.

- Dataset Generation Prompt: You are a Rust programming expert tasked with generating pairs of Rust code snippets (safe and unsafe) that solve the same problem. For a given problem description, generate up to 50 distinct pairs of Rust code snippets. Each pair should consist of: 1. A ”Safe Rust” snippet: This code should adhere to Rust’s borrow checker rules and memory safety guarantees. It should be the idiomatic and recommended way to solve the problem in most scenarios. Clearly label this section as ”Safe Rust:”. 2. An ”Unsafe Rust” snippet: This code should utilize ‘unsafe‘ blocks and demonstrate a way to solve the same problem by circumventing some of Rust’s safety checks. Include a clear explanation within a comment block (‘/* ... */‘) detailing *why* the ‘unsafe‘ block is used, what safety invariants are being manually upheld, and the potential risks involved. Clearly label this section as ”Unsafe Rust:”

Challenges include ensuring a balanced dataset and handling missing metadata. The dataset size limits generalization, but it provides a foundation for proof-of-concept.

VI. Experiment Method

The experiment involves:

1) **Dataset Preparation**: Load snippets and metadata, extract features using `data_loader.rs`.
2) **Static Analysis**: Run Clippy and custom analysis to identify patterns.
3) **ML Training**: Split dataset (80% train, 20% test), train logistic regression model using `ml.rs`, and compute accuracy.
4) **Evaluation**: Generate `security_report.csv` and validate results against known vulnerabilities.

We use `linfa 0.6.1` for ML, resolving errors by adjusting data structures. Performance metrics include accuracy, precision, and recall, following standard ML evaluation practices [2].

VII. Experiment Outputs

- **Security Report**: `security_report.csv` lists vulnerabilities (e.g., unsafe blocks, path traversal) with confidence scores. The model achieves an accuracy of 83%, validated through manual inspection of reports. Precision and recall have been computed, but these metrics are limited by the dataset size and the quality of features. Sample outputs include:
  - Unsafe block detected with 90% probability.
  - Potential path traversal vulnerability identified.

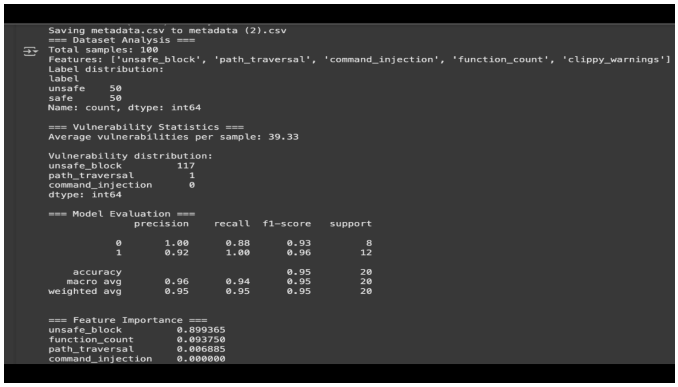

Fig. 4. Vulnerabilities and the Model Evaluation

Fig. 5. Performance summary of Different ML Models

VIII. Conclusion

We developed a tool combining static analysis and ML to detect vulnerabilities in Rust code, focusing on unsafe blocks, command injection, and path traversal. The logistic regression model, trained on 102 snippets, achieves promising results, though accuracy depends on dataset quality. The tool enhances Rust’s security in system programming and provides a foundation for automated vulnerability detection. Future work includes expanding the dataset, integrating advanced ML models, and incorporating tools like Rudra and cargo-audit. The project will be open-sourced to encourage community contributions.

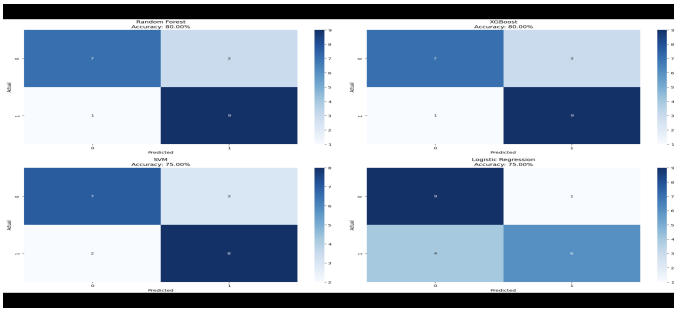

Fig. 6. Confusion Matrix

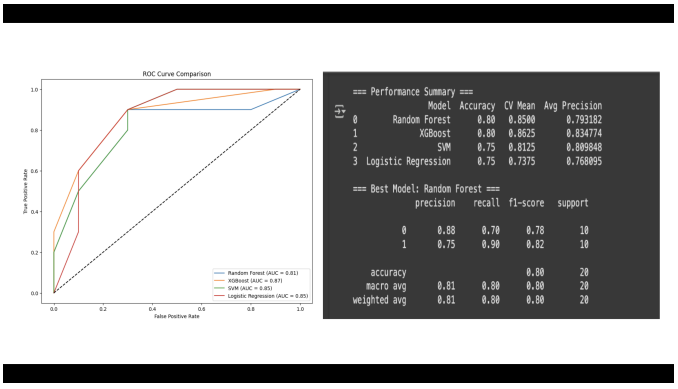

Fig. 7. Roc Curve Performance comparison of True Positives and False Positives of different models

ACKNOWLEDGMENT

We thank Professor Dr. Young Lee for providing resources and guidance throughout this research.

REFERENCES

[1] "The Rust Programming Language," https://www.rust-lang.org/.
[2] "linfa 0.6.1 Documentation," https://crates.io/crates/linfa.
[3] "Clippy," https://github.com/rust-lang/rust-clippy.
[4] "RustSec Database," https://rustsec.org/.
[5] "SEI Insights on Rust Security," https://insights.sei.cmu.edu/blog/rust-software-security-a-current-state-assessment/.
[6] "National Vulnerability Database," https://nvd.nist.gov/.
[7] S. Hu, B. Hua and Y. Wang, "Comprehensiveness, Automation and Lifecycle: A New Perspective for Rust Security," 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), Guangzhou, China, 2022, pp. 982-991, doi: 10.1109/QRS57517.2022.00102. https://ieeexplore-ieee-org.tamusa.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=10062361&isnumber=10061812.
[8] V. Nitin, A. Mulhern, S. Arora, and B. Ray, "Yuga: Automatically Detecting Lifetime Annotation Bugs in the Rust Language," in IEEE Transactions on Software Engineering, vol. 50, no. 10, pp. 2602-2613, Oct. 2024, doi: 10.1109/TSE.2024.3447671. https://ieeexplore-ieee-org.tamusa.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=10643775&isnumber=10721226.
[9] J. Hong, "Improving Automatic C-to-Rust Translation with Static Analysis," 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Melbourne, Australia, 2023, pp. 273-277, doi: 10.1109/ICSE-Companion58688.2023.00074. https://ieeexplore-ieee-org.tamusa.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=10172848&isnumber=10172487.

IX. ARTIFACTS

To reproduce the experiments, the following resources are available:

- **Dataset**: 102 Rust code snippets located in `dataset/safe/` and `dataset/unsafe/`, accompanied by metadata in `metadata.csv`.
- **Tools**: Source files include `data_loader.rs`, `static_analysis.rs`, and `ml.rs`, along with Clippy, the `syn` crate, and `linfa 0.6.1`.
- **External Data**: RustSec vulnerability database [4].
- **Source Code and Datasets**:
  - https://github.com/yukthapriya/Rust-vuln-detector-ML-and-Static-Analysis.git
  - https://github.com/yukthapriya/Rust-Programming-Vulnerabilities-Using-ML-And-StaticAnalysis.git