CS 3551: Project Presentation

Unsafe Rust in Different Domains

Abdulaziz Alotaibi

University of Pittsburgh

Computer Science Department

School of Computing and Information

Project Goal

- Understand and mitigate risks in unsafe Rust code by detecting and addressing memory bugs.
- Extend Rudra's application to analyze developer-written code across different domains, beyond just Rust's official packages.
- Apply the analysis to diverse domains (e.g., distributed systems, embedded systems, database systems).
- Help the Rust community identify and mitigate memory safety vulnerabilities in unsafe Rust.

Overview of Rudra

- **Static analysis tool for Rust:** Detects memory safety bugs in unsafe Rust code.
- **Focuses on Rust's official packages:** Designed to scale and analyze large-scale Rust ecosystems.
- **Key strengths:** Efficient at pinpointing vulnerabilities in unsafe blocks with minimal false positives.
- **Foundation for this project:** Extended Rudra to analyze developer-written code across diverse domains.

Approach and Methodology

Selecting Projects:

- Identified popular open-source Rust projects across four different domains (e.g., distributed systems, embedded systems) using GitHub.

Detecting Unsafe Rust:

- Explored tools like Carrot and Geiger to identify unsafe Rust but found them ineffective for large projects with complex dependencies.
- Developed a custom Python tool to accurately detect unsafe Rust code in these projects.

Approach and Methodology

Analyzing Unsafe Patterns:

- Studied the detected unsafe Rust code to identify common patterns and categorized them into meaningful groups.
- Prioritized the most critical patterns for deeper analysis based on their potential impact and frequency.

Isolating and Testing:

- Extracted unsafe Rust code, isolated it from dependencies, and analyzed it using Rudra to detect potential memory bugs.

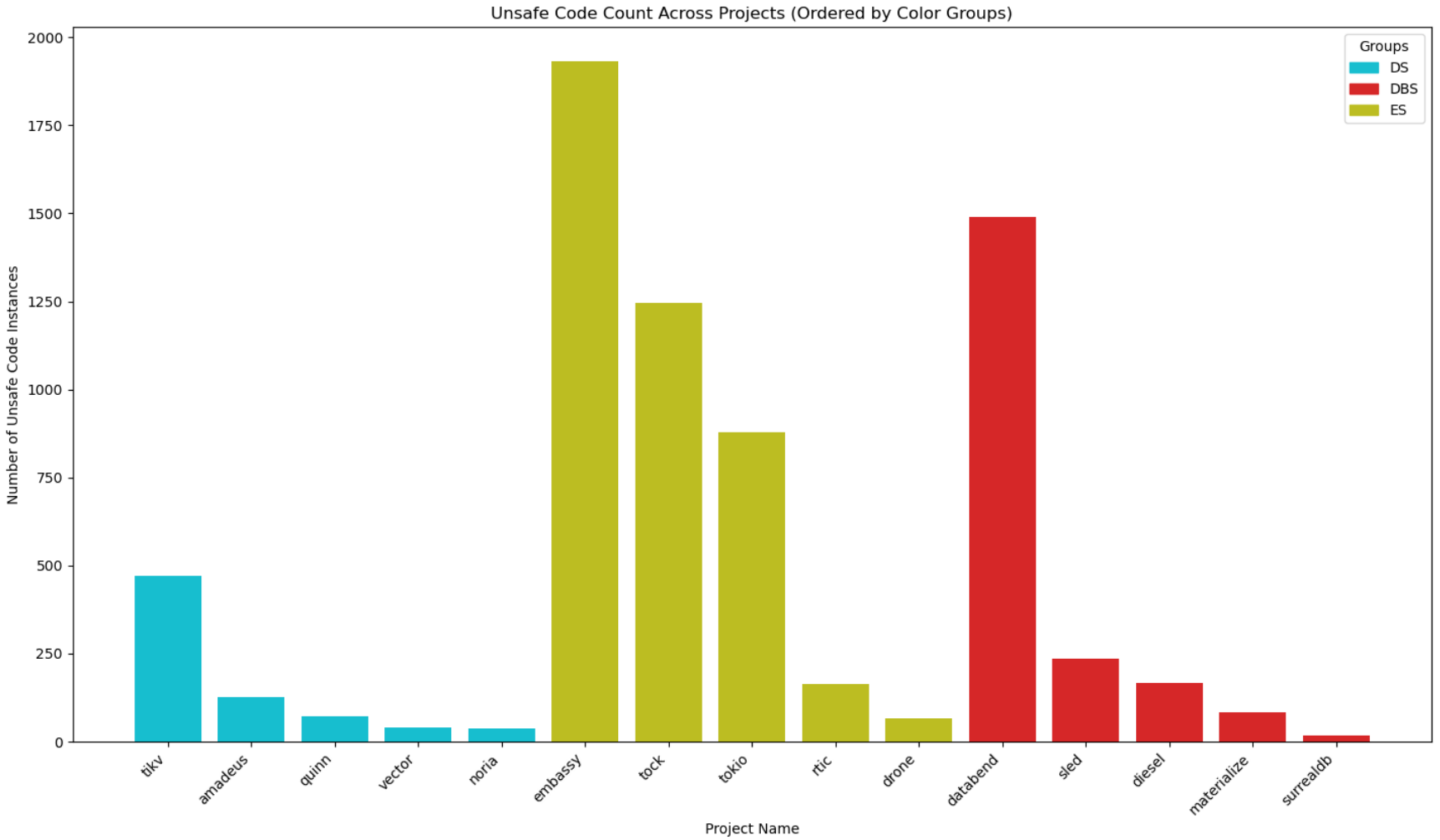
Different Domains

Distributed Systems	Embedded Systems	Database Systems	BFT
Linkerd2	Drone	Databend	AlephBFT
Noria	Embassy	Diesel	Diem
Quinn	RTIC	Materialize	Hbbft
TiKV	Tock	Sled	HotStuff
Vector	Tokio	SurrealDB	Tendermint

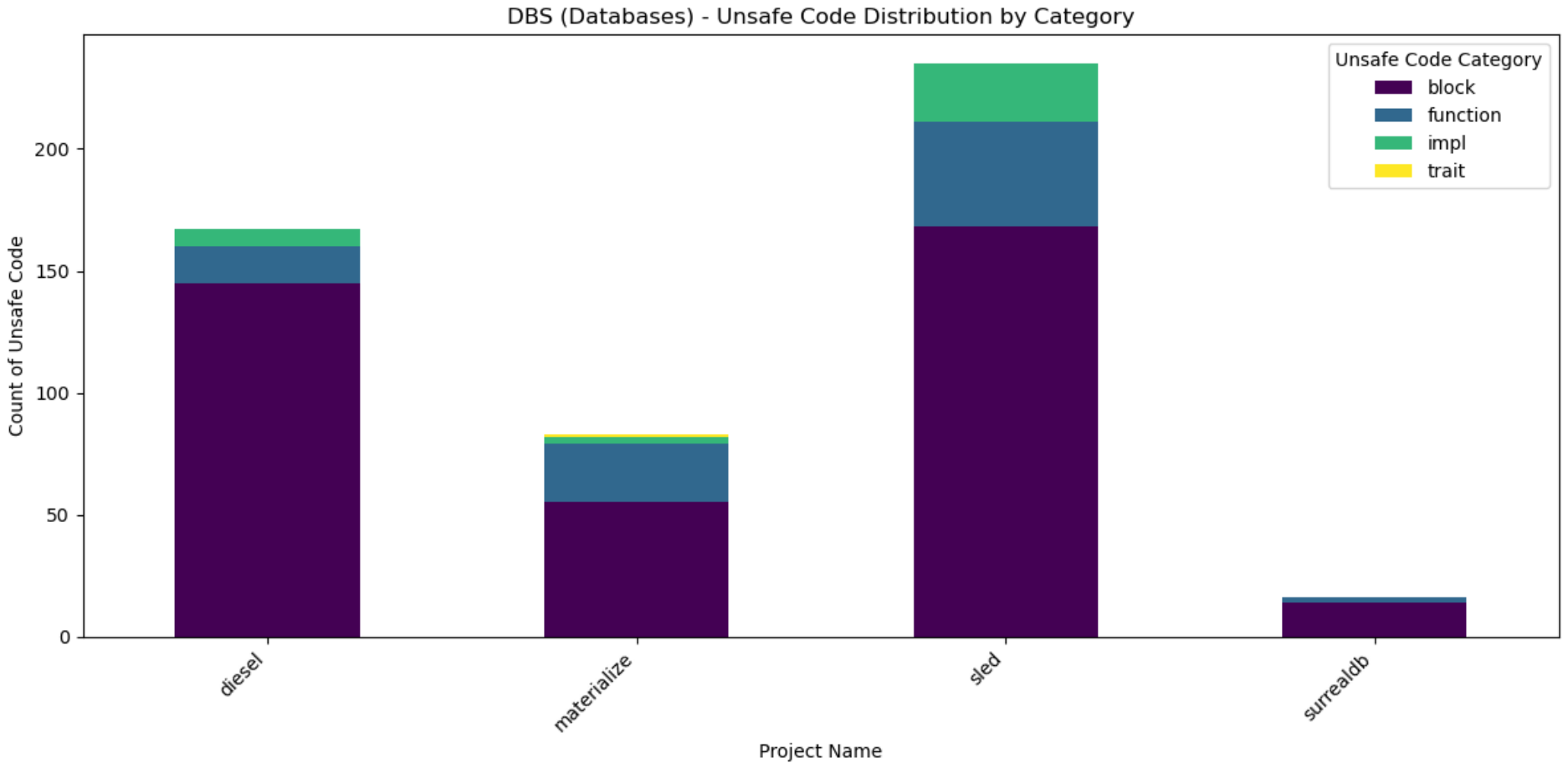
Key Results - Overview

- Unsafe Rust detection across projects in different domains.
- Analysis of unsafe Rust structures: blocks, functions, traits, and implementations.
- Categorization of unsafe Rust patterns and their significance.

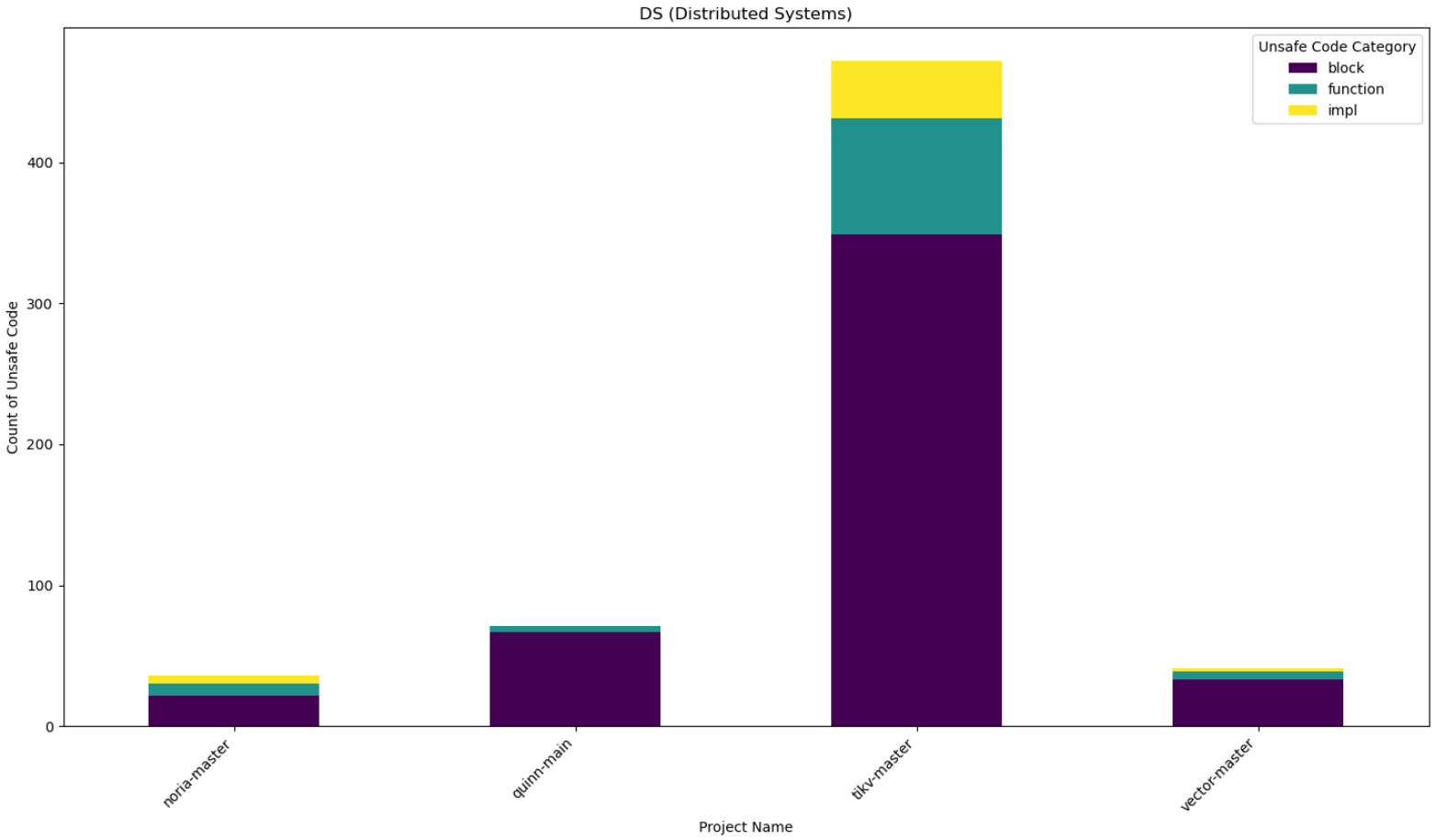
Key Results - Unsafe Rust Across Domains



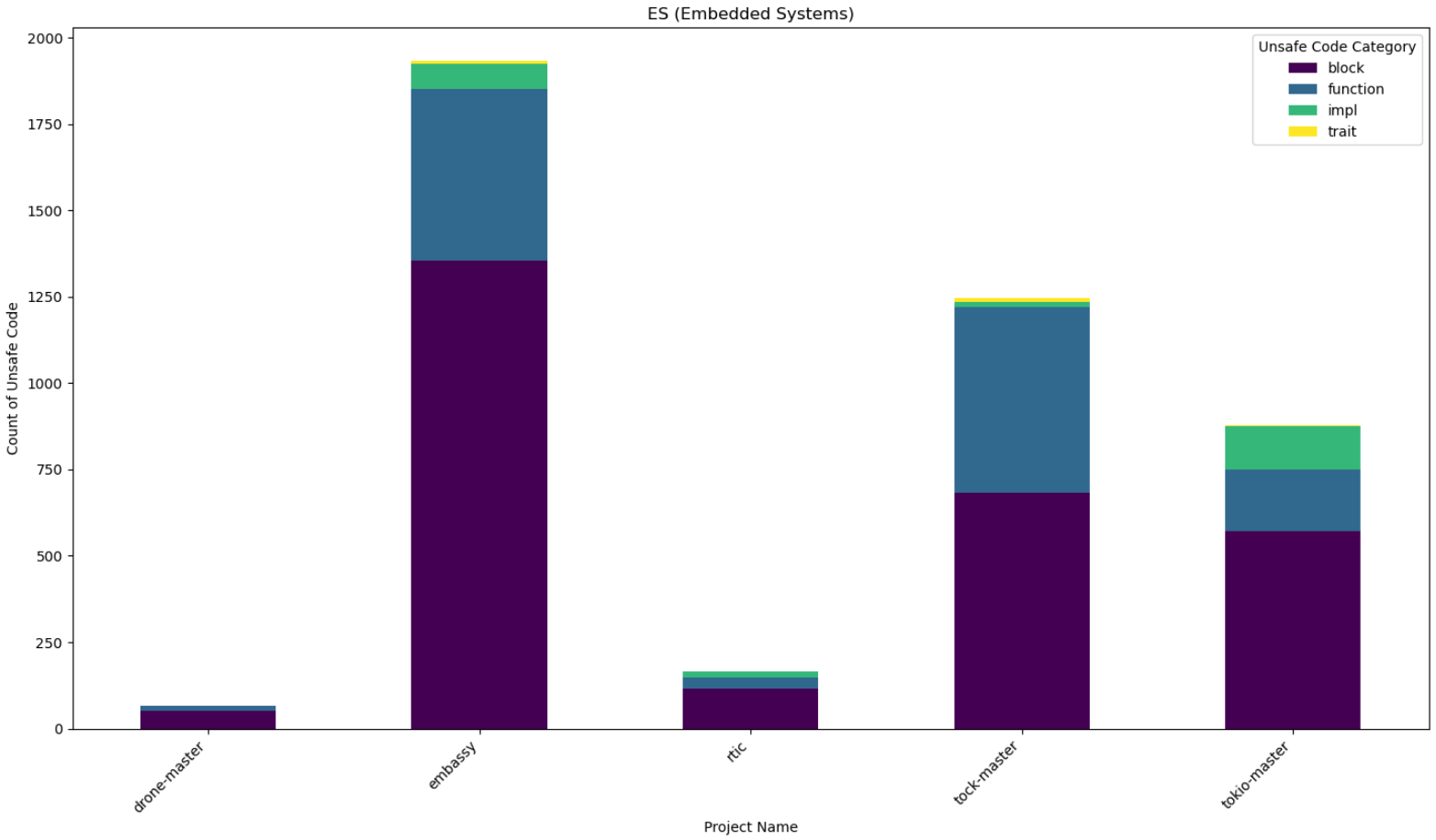
Key Results - Unsafe Rust Structures



Key Results - Unsafe Rust Structures



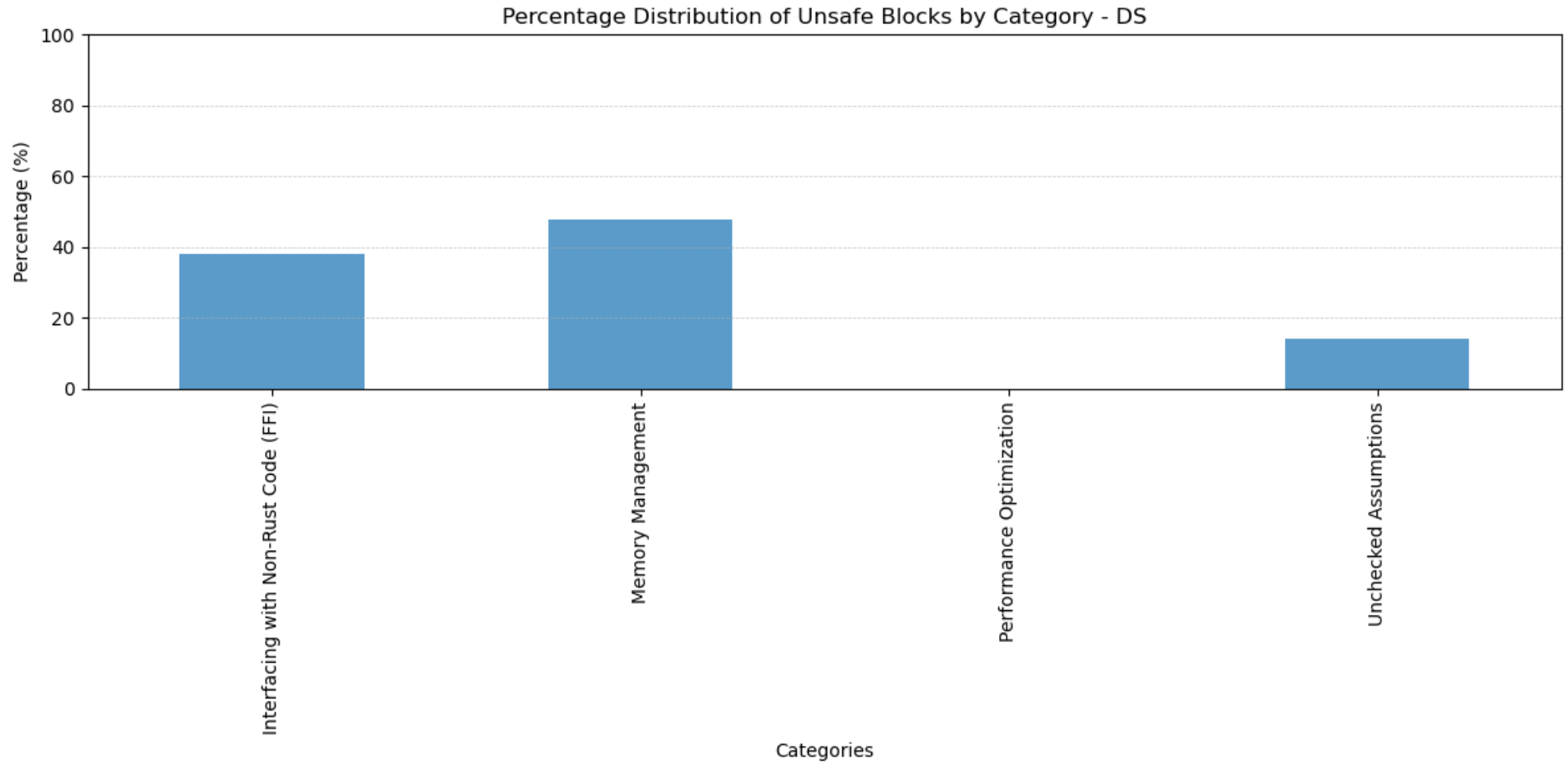
Key Results - Unsafe Rust Structures



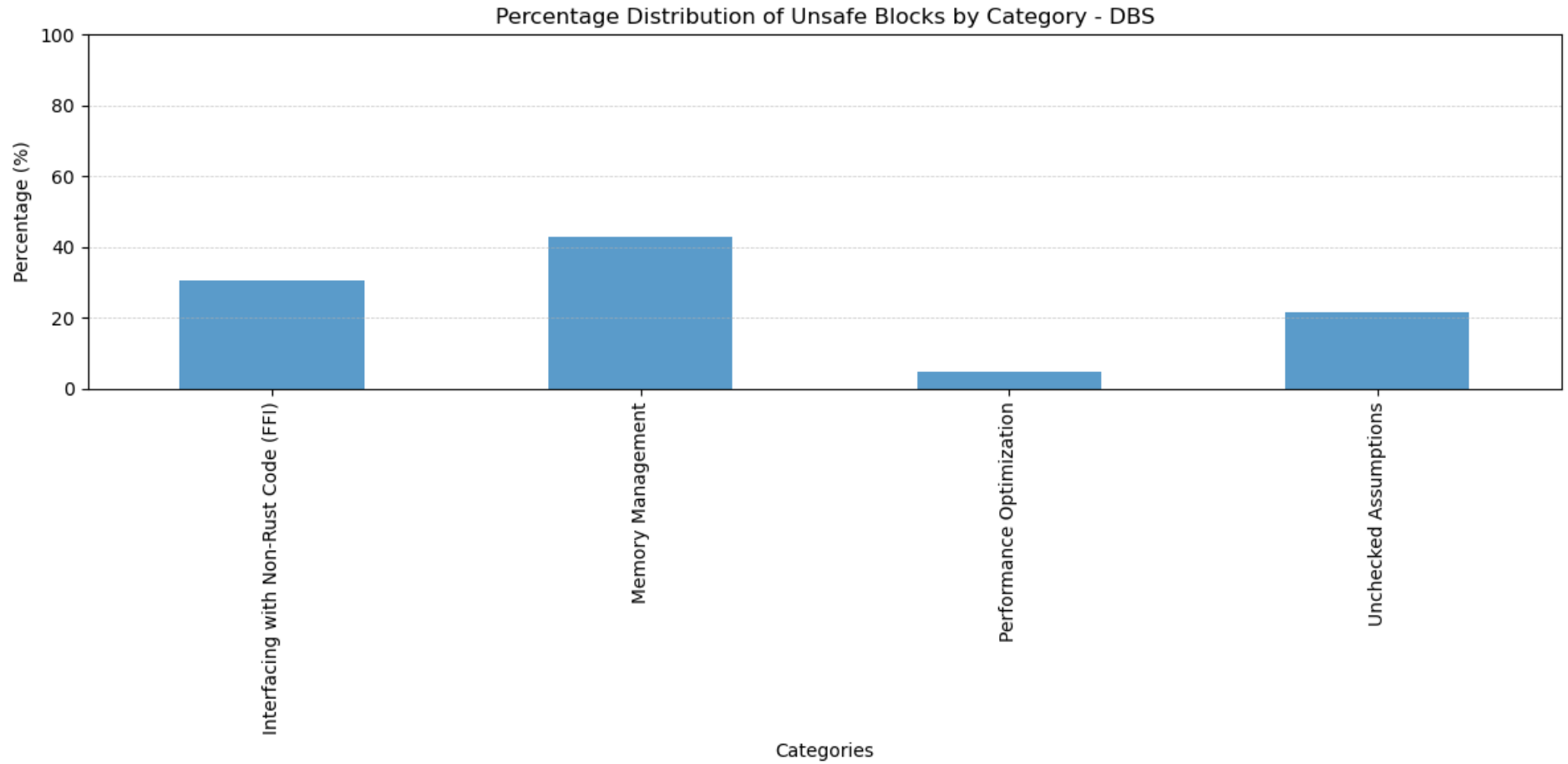
Key Results - Patterns in Unsafe Rust

- **Categories Identified:**
  - **Memory Management:** Unsafe usage for manual memory allocation and deallocation.
  - **Performance Optimization:** Unsafe blocks introduced for low-level optimizations.
  - **Interfacing with Non-Rust Code (FFI):** Unsafe code for interacting with external systems or C libraries.
  - **Unchecked Assumptions:** Unsafe blocks with unverified constraints or logic.

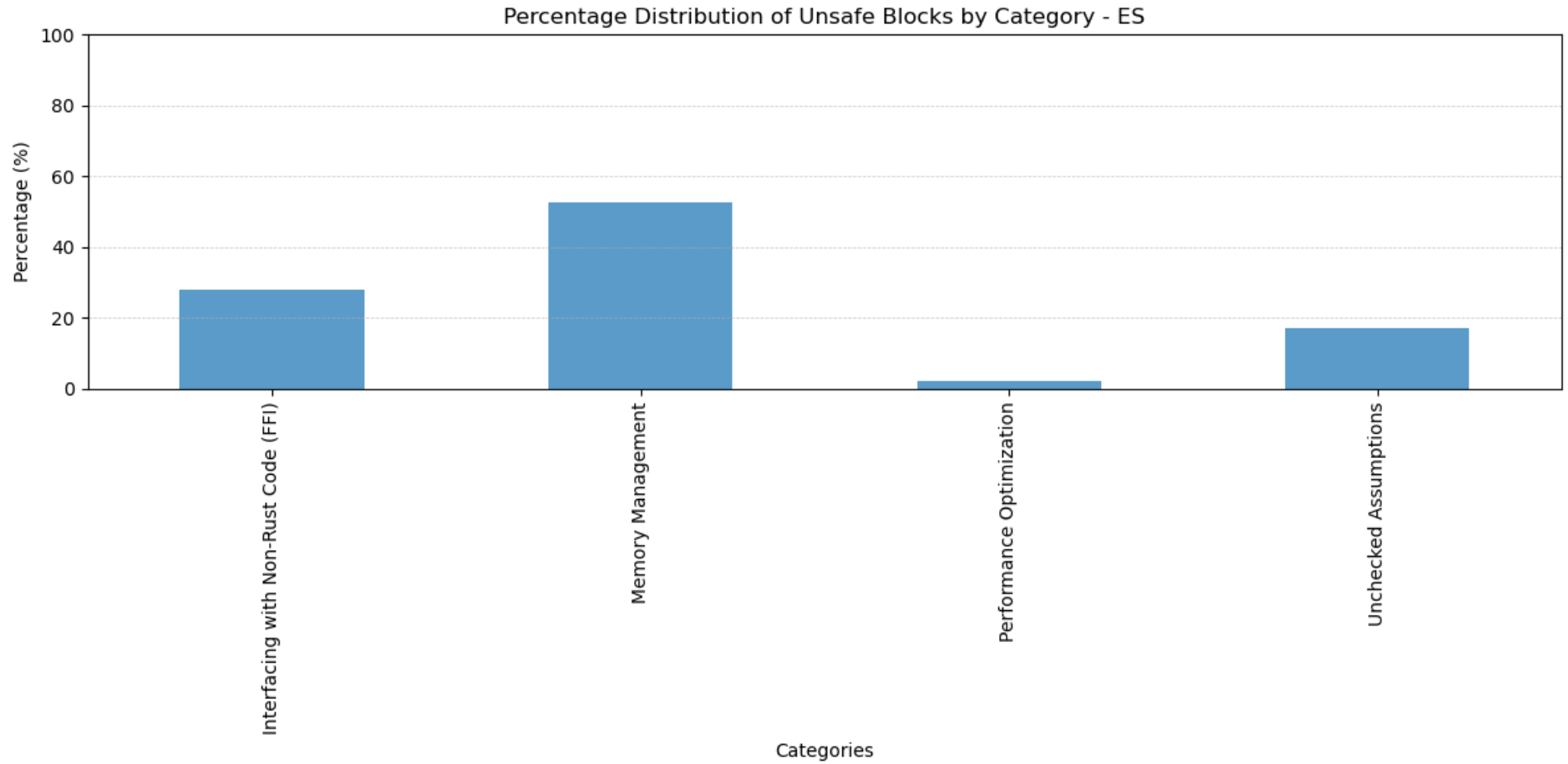
Key Results - Patterns in Unsafe Rust



Key Results - Patterns in Unsafe Rust

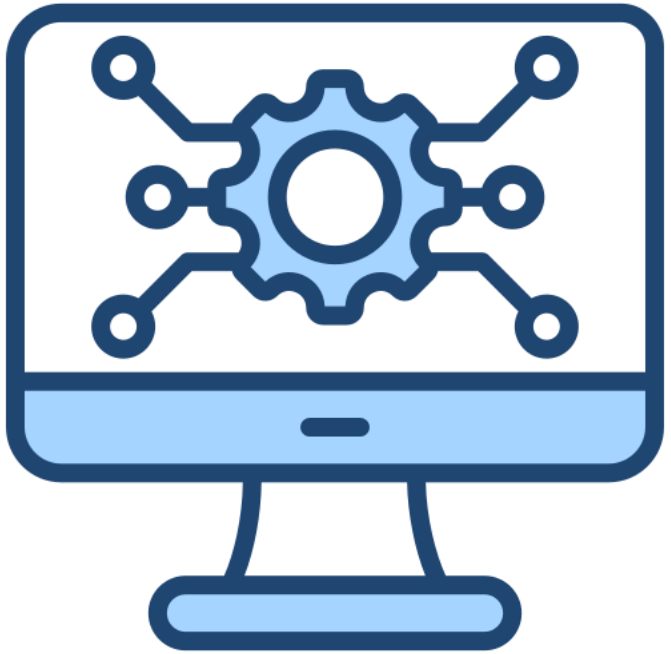


Key Results - Patterns in Unsafe Rust

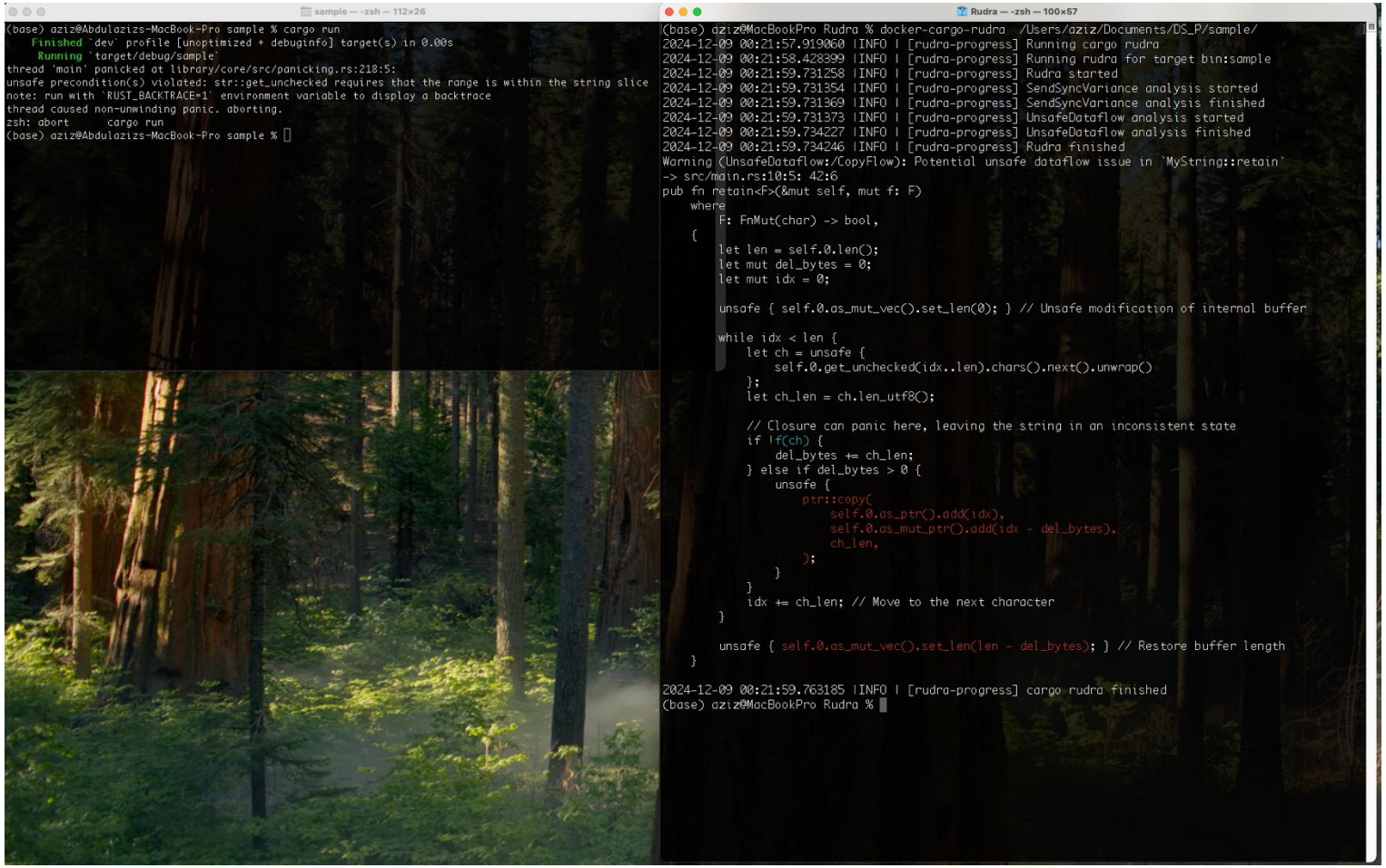


Key Results - Rudra Analysis

- Selection of Unsafe Rust
- Isolation Process
- Analysis with Rudra



Sample Analysis



Key Results - Rudra Analysis

Sample Size:

- Analyzed **5 unsafe Rust blocks** from each project.
- Analyzed a total of 60 unsafe Rust blocks: 20 blocks per domain across 3 domains.

Analysis Outcome:

- Memory Bugs Detected: None.
- Rudra flagged no memory safety issues in the selected unsafe Rust samples

Lessons Learned

- Learning Rust
- The Role of Unsafe Rust in Real-World Applications
- Capabilities and Limitations of Static Analysis Tools.
- The Complexity of Analyzing Real-World Projects
- As I’ve observed during this project, unsafe Rust is powerful but requires thorough analysis to ensure safety across diverse applications.

Q&A