Automated Language-Agnostic Multi-Metric Security Evaluation for Transpilation Workflows

Charles Lohest[1], Esteban Aguililla Klein[2], Remi Gunsett[1], Eric Viseur[1], Ken Hasselmann[3], and Antonio Paolillo[2]

[1] Thales, Tubize, Belgium
[2] Vrije Universiteit Brussel, Brussels, Belgium
[3] Royal Military Academy, Brussels, Belgium

Abstract. The migration of legacy software toward memory-safe programming languages has emerged as a promising approach to reducing vulnerability exposure in critical systems. However, rigorously assessing whether such migrations improve security remains challenging, as cross-language comparisons are affected by differences in tool maturity, code structure, and analysis coverage.

This paper presents a language-agnostic, multi-metric methodology for security evaluation in transpilation workflows. The approach combines static analysis, coverage-guided fuzzing, a dedicated metric for memory vulnerabilities, and a delta metric capturing vulnerabilities introduced and removed during transformation. To enable sound comparison, raw findings are normalized into risk-density measures by relating static analysis results to the number of functions and dynamic findings to the number of executed functions, and then mapped to bounded risk indicators using an exponential transformation. These indicators are aggregated into a global security score that quantifies relative improvement between baseline and migrated implementations.

To improve scalability, the dynamic pipeline integrates an automated, ML-assisted crash-triage component to classify fuzzing results. Experiments with controlled CWE-based implementations show that the proposed methodology yields interpretable, bias-reduced comparisons between C and Rust programs. The framework provides a reproducible and extensible approach for quantifying security gains in transpilation and language migration processes. This controlled setting offers an initial validation, and future work will extend the methodology to industrial-scale codebases.

Keywords: Software Security · Transpilation · Memory Safety · Rust Language · Security Evaluation · Machine Learning

1 Introduction

Software security is a critical concern in increasingly interconnected systems. Ensuring the reliability and resilience of software components is now a priority for organizations seeking to reduce the attack surface and exposure to cyber threats.

To support this, organizations increasingly rely on automated assessment pipelines that combine static analysis (e.g., SonarQube [37], Semgrep [36]) with dynamic techniques such as coverage-guided fuzzing [25]. Each provides incomplete evidence on its own due to false positives and limited coverage [30]. In parallel, recent guidance from major institutions recommends migrating legacy components to memory-safe languages, often via transpilation (e.g., C/C++ to Rust) [5,28,39]. However, current evaluation practice still emphasizes performance, maintainability, and functional equivalence, leaving open a key question: *how can we fairly and repeatably determine whether the migrated implementation is actually more secure?*

We address this by proposing a language-agnostic, multi-metric methodology for comparing the security posture of observationally equivalent implementations, illustrated on C/C++ and Rust. Our approach combines static analysis, coverage-guided fuzzing, a memory-focused metric, and a delta metric that captures vulnerabilities introduced and removed during transformation. To improve scalability, the dynamic stage incorporates a machine-learning-assisted crash-triage subsystem that classifies AFL++ artifacts and replay traces.

To enable comparison across languages and tools, raw findings are normalized into risk-density measures based on semantic and executed exposure and mapped to bounded risk indicators $[0, 1]$ through a calibrated transformation.

This work is motivated by a research initiative to evaluate transpilation tools [13,2,8], which translate and compile source code from one language into an equivalent implementation in another. Our main case study is translating C and C++, valued for performance and flexibility but prone to memory-safety vulnerabilities, into Rust, a modern language with strong memory-safety guarantees. Although we focus on C/C++-to-Rust, the methodology is language-agnostic and applicable to other cross-language security comparisons.

We evaluate the methodology on a dataset of controlled CWE-based implementations. This controlled setting enables reproducible experiments, isolates the impact of language features, and avoids confounding effects from large-scale architectural complexity. It thus provides a baseline for analyzing the methodology before extending it to real-world systems.

Section 2 reviews language migration, transpilation, and security measurement work, emphasizing the lack of rigorous methods for fair cross-language security comparison. Section 3 defines the security comparison problem, including objectives, challenges, and analysis tools used to reduce tool-induced bias. Section 4 describes the scoring process, including normalization and calibration to ensure fairness. Section 5 reports initial findings on CWE-based samples. Finally, Section 6 provides a summary and outlines future directions, including industrial applications.

2 Related Work

Recent research has explored migrating legacy systems written in C and C++ to memory-safe languages, with particular attention to Rust-based transpilation

approaches [5,11]. These works investigate automated translation techniques and address the technical challenges involved in preserving functional equivalence across languages.

Most existing efforts emphasize the theoretical guarantees of Rust's ownership and borrowing model, which, by construction, eliminate broad classes of memory-safety vulnerabilities. However, ensuring the absence of vulnerabilities in practice remains inherently difficult [20,32].

Beyond tool development, several studies aim to empirically demonstrate the security benefits of migrating from unsafe to safer languages. These works typically rely on vulnerability counts or crash rates without controlling for biases introduced by language differences, tool maturity, or detection capabilities. As a result, reported improvements may reflect analysis artifacts rather than intrinsic security gains. Addressing this limitation is a central goal of our normalization strategy [26,23].

In parallel, major institutions and organizations, including the White House and large technology companies such as Google, have publicly advocated for the adoption of memory-safe languages in critical software systems [19,28,39,18]. These initiatives often report significant reductions in memory-related vulnerabilities following the introduction of Rust into existing codebases. While these reports provide valuable high-level evidence of security improvements, they typically lack a detailed, reproducible methodology for measuring, normalizing, and validating security gains. In particular, they rarely explain how introduced vulnerabilities are accounted for or how improvements are distinguished from shifts in tooling or reporting practices [12,16].

In addition to traditional static and dynamic analysis techniques, recent research has explored alternative approaches to vulnerability detection and security assessment. These include advances in coverage-guided fuzzing, hybrid analysis techniques, and the use of large language models (LLMs) [14,9,21,22,15] for vulnerability identification and code auditing. While these approaches improve detection capabilities, their primary focus remains on identifying vulnerabilities within a single codebase rather than on enabling systematic, unbiased comparisons across programming languages [4,27]. In contrast, the machine-learning component introduced in this work supports scalable crash triage in fuzzing-based evaluation, assisting the classification of dynamic failures without replacing traditional static or dynamic analysis techniques.

Overall, prior work motivates memory-safe migration but rarely provides an end-to-end, language-agnostic methodology that combines complementary evidence sources and normalizes heterogeneous tool outputs into comparable, interpretable metrics for cross-language security evaluation [31,7,41]. Our work targets this methodological gap for transpilation and migration workflows.

3 Evaluation of Security

Comparing the security posture of C/C++ and Rust implementations requires careful control of methodological bias. Differences in language design, memory

models, and tool maturity can distort raw vulnerability counts, making direct comparisons unreliable. In particular, the more mature analysis ecosystem for C/C++ tends to produce more alarms, many of which are false positives, so higher detection counts do not necessarily reflect higher intrinsic risk [30].

To address these challenges, we propose a structured, repeatable methodology to quantify security gains throughout transpilation.

3.1 Objectives

The objective of the security evaluation is to determine whether the transpilation process yields a measurable and meaningful reduction in security risks associated with legacy C/C++ codebases. Since Rust integrates compile-time guarantees that eliminate an entire class of memory vulnerabilities, the expectation is that the resulting software should inherently be more secure.

To validate this hypothesis, our evaluation seeks to quantify differences in vulnerability exposure between the two languages using a set of vulnerability detection programs; monitor how security evolves at each step of the transpilation workflow; assess whether security improvements are consistent and sustained; and verify that newly introduced vulnerabilities (if existing) do not outweigh those eliminated.

3.2 Challenges

A direct comparison of security properties between C/C++ and Rust raises several methodological challenges. First, the languages follow fundamentally different paradigms regarding memory safety and access control, which influence how vulnerabilities manifest.

Second, the maturity and diagnostic capabilities of analysis tools are not equivalent. For example, C and C++ have been supported by industrial-grade static analysis platforms such as SonarQube for many years, benefiting from extensive rule sets, community feedback, and tuning based on real-world vulnerability patterns. In contrast, Rust support in comparable platforms is comparatively recent, and rule coverage remains less mature than for C/C++. Without appropriate controls, this can bias vulnerability counts toward the better-supported language, confounding language effects with tooling maturity.

These challenges require a robust evaluation strategy that normalizes scores, combines complementary analysis techniques, and distinguishes genuine improvements from tool-induced distortions.

3.3 Tools

A careful selection of security analysis tools has been made to ensure an objective and comprehensive evaluation of the security posture of C/C++ and Rust codebases. Since vulnerabilities can be detected using different techniques, the toolset is divided into two complementary categories: static and dynamic analysis.

Static Analysis. Static analysis introduces a significant risk of bias because tooling maturity differs substantially between languages. For C and C++, vulnerability detection is supported by well-established platforms such as SonarQube and Semgrep [37,36]. These tools rely on decades of rule development and broad industrial adoption, providing comprehensive and reliable diagnostics.

In contrast, Rust support in comparable platforms is more recent. For example, official Rust integration in SonarQube was introduced only in April 2025 [38], and the available security rules remain fewer and less mature than those for C/C++. Using these tools alone can bias the evaluation toward the better-supported language, reflecting differences in tooling rather than intrinsic language properties.

To mitigate this issue, additional Rust-specific static analyzers are incorporated to improve diagnostic coverage. **Rudra** is a memory-safety-oriented static analyzer tailored to Rust's ownership and borrowing model, detecting misuse patterns linked to concurrency, aliasing, and lifetime errors [33]. **Cargo Geiger** reports the presence and propagation of `unsafe` blocks and dependencies; although `unsafe` constructs may be justified for low-level operations, their occurrence highlights security-sensitive regions of the code [6]. **Clippy**, the official Rust linter, is configured with strict settings to flag non-idiomatic, risky, or error-prone coding constructs that may reduce long-term safety and reliability [35].

When combined with SonarQube, this Rust toolset increases diagnostic coverage where general-purpose scanners lag. Because unequal tool maturity can dominate raw counts, we treat tool outputs as noisy measurements rather than ground truth and enforce comparability through: (i) a shared language-agnostic taxonomy for mapping findings, (ii) severity-and-category weighting applied uniformly, and (iii) normalization into risk densities to reduce structural and reporting-volume effects. To avoid introducing a counter-bias, we restrict the Rust-specific checks and tools to equivalent detection mechanisms in the C/C++ baseline (e.g., memory safety and unsafe exposure).

The goal is to compensate for differences in tooling maturity and achieve comparable analytical coverage across languages, ensuring that results reflect language security level rather than tool support.

Dynamic Analysis. Static analysis alone cannot capture all vulnerability classes, especially those triggered only at runtime, such as memory corruption, unexpected state transitions, or improper input handling. To complement static inspection, we perform dynamic analysis using automated fuzzing. We employ the AFL++ fuzzer [1] for its strong performance in detecting memory-related issues and crash-inducing behavior in native binaries. AFL++ natively supports instrumentation for C and C++, enabling efficient detection of runtime violations, including buffer overflows, heap corruption, and illegal memory accesses. Crucially, AFL++ also provides an equivalent Rust integration mode [34], allowing Rust programs to be instrumented and fuzzed with the same mutation strategies and runtime feedback metrics. This ensures that the dynamic evaluation of both language versions occurs under comparable conditions, so differences

in vulnerability discovery can be attributed to language properties rather than tooling. During fuzzing, inputs are automatically generated and mutated from an initial corpus to exercise diverse execution paths. When a crash occurs, the triggering input is saved for replay and analyzed using debugging tools, along with an assisted triage stage that automates vulnerability classification and severity assessment. Dynamic analysis thus offers essential insight into runtime robustness, particularly memory safety, and complements static evaluation to provide a more comprehensive view of the security posture of both implementations.

Mitigating Tool-Induced Bias. The evaluation relies on language-specific analysis tools because the tools used for the different languages have different scoring systems and way to classify findings. To mitigate discrepancies, their findings are mapped to a common set of vulnerability categories (e.g., CWE), enabling consistent comparison despite naming or reporting differences.

Vulnerability counts are then converted into risk-density metrics (e.g., per function or execution unit), which temper the impact of tool-specific detection volume. Tools with different capabilities may produce different absolute counts, but normalization expresses findings relative to structural or executed exposure, shifting the focus from "how many issues are reported" to "how densely issues occur".

The methodology restricts evaluation to vulnerability categories that can be reliably identified across tools (e.g., shared CWE classes) and applies identical scoring and aggregation rules across languages. Vulnerabilities unique to a single toolchain do not skew the final score, and comparable issues are treated uniformly, so the comparison centers on shared, consistently represented vulnerability types.

These measures lessen, but do not eliminate, tool-induced bias. The resulting indicators should be interpreted as relative measures within a controlled, functionally aligned setup rather than as absolute, tool-independent security metrics.

4 Methodology

This section describes how vulnerability findings are processed, measured, and normalized into comparable security metrics for the original C/C++ implementation and its transpiled Rust equivalent.

Because the two languages differ significantly in both inherent safety guarantees and maturity of analysis tools, a direct one-to-one comparison of raw vulnerability counts would lead to misleading conclusions. Instead, the methodology relies on a structured scoring approach that combines multiple detection techniques, severity weighting, and normalization formulas. This enables the evaluation to capture the actual security impact of transpilation rather than the uneven diagnostic capabilities of tools.

The evaluation is divided into four complementary sub-metrics: static analysis evaluation (**30%** of the global score), dynamic analysis evaluation (**45%**),

occurrence of memory-related vulnerabilities (**10%**), and the net variation of vulnerabilities introduced vs. removed during transpilation (**15%**).

These weights reflect the relative evidential strength and security impact of each metric: dynamic analysis is emphasized due to exploitable runtime failures, while static analysis captures broader structural weaknesses. The memory-specific and delta metrics refine the evaluation by isolating memory exposure and accounting for vulnerabilities removed versus introduced during transpilation. The weighting scheme is designed to balance complementary sources of evidence while preventing any single metric from disproportionately influencing the overall security assessment.

4.1 Normalization

At a basic level, the score of each sub-metric could be computed as a raw score by applying a weight and a severity factor to each detected vulnerability, using the following formula:

$$S_{\text{raw}} = \sum (w_{\text{type}} \cdot s_{\text{severity}}).$$

However, raw scores are not comparable across languages because they are strongly affected by code structure, size, and tool-specific coverage. As discussed in Section 3.2, analysis tools differ in rule sets and maturity across languages, so raw scores can conflate security risk with measurement artifacts [3,24,42].

To address these limitations and ensure a fair comparison, we introduce a normalization process. This process is decomposed into several steps, each described in the following subsections.

Vulnerability Classification. The first step is to classify detected vulnerabilities into language-agnostic categories and assign each category a weight reflecting its potential security impact. Based on the CWE abstractions, we define the following vulnerability classes and their associated weights (inside the parentheses). Those weights have been attributed based on the Common Vulnerability Scoring System (CVSS) [17]: Memory safety (8), Injection (6), Authentication/Authorization (6), Concurrency (3), Cryptography/Secrets (3), and Logic/State (1).

These categories are applicable across programming languages and allow heterogeneous tool outputs to be mapped onto a common semantic basis. Each detected vulnerability is assigned to exactly one category, ensuring consistency in scoring.

In addition, we incorporate a severity factor also based on CVSS to distinguish vulnerabilities within the same category: Critical (1.0), High (0.75), Medium (0.4), and Low (0.1).

The impact of a vulnerability is therefore computed as:

$$\text{impact} = w_{\text{type}} \cdot s_{\text{severity}}$$

This prevents a large number of low-severity issues from outweighing a smaller number of high-impact vulnerabilities.

Static Analysis Normalization. Static analysis evaluates *potential* vulnerabilities across the entire codebase, yielding a raw static score ($S_{\text{static,raw}}$) that reflects the total potential risk but remains highly sensitive to code size and structure. To obtain a fair metric, this raw score must be transformed into a *risk density*.

This is achieved by normalizing the raw static score by a semantic exposure metric. Possible denominators include control-flow graph (CFG) nodes or basic blocks, the number of functions, or lines of code (LOC).

All of these metrics are valid representations of code exposure. However, in this study, we select the **number of functions** to integrate the impact of each function into the security evaluation. The normalized static score is therefore computed as:

$$S_{\text{static,norm}} = \frac{S_{\text{static,raw}}}{number\_of\_functions}.$$

This normalization converts total potential risk into a potential risk density per unit of logic, mitigating biases introduced by language verbosity or refactoring.

Dynamic Analysis Normalization. Dynamic analysis differs fundamentally from static analysis because it observes vulnerabilities only along executed paths. Consequently, normalizing by total code size or the total number of functions is inappropriate, as unexecuted code does not contribute to observed runtime risk. Instead, dynamic normalization relies on an *executed exposure* metric, such as covered CFG edges or executed functions.

To remain consistent with the static normalization strategy, we choose the number of **executed functions** as the denominator. The normalized dynamic score is computed as:

$$S_{\text{dynamic,norm}} = \frac{S_{\text{dynamic,raw}}}{executed\_functions}.$$

Bounding and Calibration. After normalization, we obtain two metrics: the potential risk density $S_{\text{static,norm}}$ and the observed risk density $S_{\text{dynamic,norm}}$.

However, those metrics do not share a common numerical scale and cannot be aggregated directly. To address this, we apply a bounding transformation that maps normalized scores to an interpretable risk indicator in the interval $[0, 1]$. This produces a common, monotonic risk scale across tools and languages, enabling aggregation without mixing incompatible units.

Following established practices in quantitative risk modeling, we use an exponential bounding function [10]:

$$R = 1 - e^{-kS},$$

where S denotes a normalized score and k is a calibration parameter. Separate values are used for static and dynamic analyses: k_{static} is chosen smaller to reflect

uncertainty and false positives in static analysis, while k_{dynamic} is chosen larger to reflect the stronger evidential value of observed runtime failures.

This function has two key properties. First, it is monotonic, preserving score ordering ($S_1 > S_2 \Rightarrow R_1 > R_2$). Second, it is bounded in $[0, 1]$, enabling the definition of interpretable risk thresholds, for example $R < 0.2$ (low risk), $0.2 \leq R < 0.8$ (moderate risk), and $R \geq 0.8$ (high risk).

Additionally, the exponential form captures diminishing returns; early findings increase risk sharply, while additional findings contribute progressively less.

The calibration parameter k is derived by anchoring a normalized score S_{high} to a target high-risk level R_{high}, using:

$$k = -\frac{\ln(1 - R_{\text{high}})}{S_{\text{high}}}.$$

The values of S_{high} are defined by explicit acceptance thresholds, such as unacceptable densities of high-impact memory vulnerabilities per function or per executed function. This calibration encodes risk tolerance explicitly and is applied consistently across languages.

Sensitivity and Interpretation of Design Parameters. Calibration thresholds determine how quickly the risk indicator grows as vulnerabilities accumulate. Lower thresholds trigger earlier penalties and suit safety-critical or high-assurance settings, where even a few issues are unacceptable. Higher thresholds delay risk escalation and fit environments that can tolerate some residual risk.

These parameters encode different risk models and operational constraints. Although we fix them in this study for comparability, the framework lets practitioners tune them to their context.

For instance, an organization migrating a legacy C component to Rust in a safety-critical system (e.g., embedded control software) cannot tolerate runtime failures such as memory corruption. Emphasizing dynamic analysis and using low thresholds yields a conservative risk indicator: even a small number of crashes or undefined behaviors sharply increases the score, supporting strict go/no-go decisions.

In a less critical industrial context setting (e.g., a backend service with monitoring and recovery), some residual risk is acceptable. Higher thresholds then produce a more gradual increase in the risk indicator, letting decision-makers weigh security signals against trade-offs such as maintainability or performance.

These examples show how parameter choices shape the interpretation of results and support context-dependent decisions. In this work, normalization turns raw findings into language-agnostic risk-density metrics, then maps them to bounded risk indicators via a calibrated exponential transformation, enabling interpretable aggregation in a unified evaluation methodology.

4.2 Static Analysis

Static analysis constitutes the first component of the evaluation. As discussed in Section 3.2, C/C++ and Rust require separate toolchains due to differences in tool maturity. We therefore use parallel pipelines to preserve comparability.

Static Analysis for C/C++. For the original legacy version written in C or C++, the analysis primarily relies on SonarQube. The choice of this tool is motivated by its high maturity for C/C++ codebases, stemming from decades of industrial deployment and a large, continuously updated set of vulnerability rules. SonarQube scans the entire program and produces a structured report listing security vulnerabilities, code smells, and potential bugs, each with an associated severity level.

To convert this raw output into a quantitative security score, the following steps are applied:

1. Execute a full SonarQube scan on the C/C++ codebase.
2. Extract all reported findings and classify each into a dedicated vulnerability category based on the underlying CWE type.
3. Assign a weight to each category, reflecting its security impact (e.g., memory corruption issues are weighted significantly higher than minor logic anomalies).
4. Compute a weighted security score as the sum of all vulnerability weights.
5. Apply the two-step normalization procedure (Section 4.1) to reduce detection-volume bias and obtain a comparable score.

This process yields a reliable measurement of static vulnerability exposure in the original implementation.

Static Analysis for Rust. Given the limitations of tooling maturity discussed in Section 3.2, additional Rust-specific analyzers are incorporated. Each tool produces its own report. These reports are then combined and transformed into a unified SonarQube-compatible format, enabling the Rust static analysis results to be processed using the same classification, weighting, and normalization procedures as for C/C++.

Thus, the equivalent scoring pipeline mirrors the C/C++ one: SonarQube, Rudra, Cargo Geiger, and Clippy are executed on the Rust codebase, their reported vulnerabilities are merged into a single consolidated dataset and classified into the same categories used for C/C++ scoring, and the resulting weighted security score is computed and passed through the two-step normalization.

Outcome and Weight in Global Evaluation. At the conclusion of this step, two independent, normalized static analysis scores are obtained: one for C/C++ and one for Rust, both expressed on a comparable scale. This metric contributes **30%** of the global security score. This weighting reflects the maturity and coverage of static analysis tools, while acknowledging that critical runtime issues may only be revealed through dynamic techniques.

4.3 Dynamic Analysis

Static analysis may miss vulnerabilities that manifest only at runtime. Dynamic analysis complements it through fuzz testing, which can reveal concrete failures such as memory corruption and input-triggered crashes.

Fuzzing Instrumentation. Before running a fuzzing campaign, the program must be instrumented. For C/C++, compilation is performed with the AFL++ instrumentation toolchain, which inserts lightweight tracing hooks and enables detection of memory-safety violations via sanitizers such as AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan). This allows the fuzzer not only to detect process crashes but also to detect silent memory errors, such as buffer overflows, use-after-free, or heap corruption.

Rust follows the same workflow using the Rust-compatible AFL++ mode, ensuring equivalent observability and avoiding methodological bias.

Fuzzing Campaign Execution. Once instrumented, the program is executed under AFL++, which automatically mutates inputs to explore as many execution paths as possible. The objective is to maximize code coverage and trigger unexpected states. When a crash or abnormal behavior is detected, AFL++ stores the responsible input in a dedicated crash corpus. These saved inputs represent distinct potential vulnerabilities found during fuzz execution.

Crash Reproduction and Classification. After the fuzzing campaign, all collected inputs are replayed under a debugger (e.g., `GDB` & `Rust-GDB`) to validate and analyze the observed failures. This post-processing step enables precise identification of the root cause of each crash, categorization of the vulnerability type (e.g., heap overflow, use-after-free, or stack-based issue), elimination of non-security-relevant crashes and false positives, and alignment of confirmed findings with the predefined severity scoring criteria.

ML-Assisted Crash Triage. We introduce a machine-learning-assisted crash triage stage to enable scalable, consistent classification. Each detected crash is labeled with three attributes: (i) exploitability (vulnerability vs. non-security crash), (ii) a taxonomy-based vulnerability category, and (iii) a severity level. Classification relies on evidence from AFL++ reports and debugger replay traces collected during crash reproduction.

We employ a fine-tuned large language model deployed via `Ollama` [29], trained on annotated crash reports and debugging traces. The model learns to associate execution symptoms (e.g., faulting instructions, stack traces, memory access violations) with vulnerability categories. Given an AFL++ report and corresponding debugger replay output, it decides whether the crash is exploitable and assigns the most appropriate taxonomy category.

We evaluate the triage model on a validation set of 100 diverse samples, including both vulnerable and non-vulnerable samples. Each sample is compiled

and executed with the same AFL++ instrumentation as in the main experiments, and crash traces are obtained via deterministic debugger replay. On this dataset, the system achieves about 95% classification accuracy, enabling automated prioritization and scoring of fuzzing findings while substantially reducing manual analysis.

The small validation set of 100 samples limits statistical confidence and may under-represent real-world crash diversity, providing a helpful indication rather than a definitive measure of generalization.

The model consumes structured inputs from AFL++ reports and deterministic debugger traces, thereby reducing variability and supporting consistent classification. Its role is to group discovered crashes into broader vulnerability types, not to detect vulnerabilities directly from code. Because the gap between vulnerability scores is small, misclassifications have limited impact on the overall score.

The ML-based triage is an optional accelerator that reduces analysis effort within the overall methodology, and manual or rule-based classification can be used instead.

Dynamic Security Scoring. For the dynamic analysis pipeline, each detected vulnerability category is assigned a weight proportional to its criticality in the evaluated language. A weighted dynamic score is computed and normalized using the two-step method presented in Section 4.1. This yields two independent, comparable dynamic analysis security scores: one for the original C/C++ program and one for the transpiled Rust version.

Role in Global Assessment. Dynamic analysis captures directly exploitable runtime vulnerabilities, particularly memory-safety violations. For this reason, it receives the highest contribution among evaluation components, accounting for **45%** of the total security score.

4.4 Memory-Related Vulnerabilities

When comparing C/C++ and Rust implementations, memory-related vulnerabilities are particularly important. This metric evaluates whether transpilation reduces exposure to memory-safety defects in practice.

Target Vulnerability Classes. This metric focuses exclusively on vulnerabilities that directly involve incorrect memory usage, including buffer overflows, use-after-free, and out-of-bounds accesses; invalid pointer usage (null or uninitialized); data races and unsafe memory access; and integer overflows impacting memory safety.

These categories are typically associated with the highest severity levels because they enable critical exploitation techniques such as remote code execution or arbitrary memory access.

Evaluation Method. Memory vulnerability counts are extracted from both previous evaluation stages, static analysis reports (Section 4.2), and dynamic fuzzing crash reports (Section 4.3).

The occurrences are then aggregated to produce a memory-specific score that reflects the number of memory-related issues detected, their severity, and the extent to which they remain or are eliminated after transpilation.

This metric is treated separately because memory corruption vulnerabilities are particularly important in safety-critical systems.

Weight in the Global Score. Although memory-related vulnerabilities are already accounted for with considerable weighting in the static and dynamic analysis scores, this metric provides a dedicated focus on dangerous vulnerabilities affecting C/C++ software. Therefore, it contributes **10%** to the final security score.

This weighting ensures that improvements explicitly related to Rust's memory safety model are properly highlighted without disproportionately overshadowing other vulnerability dimensions.

4.5 Computing a Delta

The delta metric evaluates the net impact of the transpilation process on security. While static and dynamic analyses quantify vulnerabilities in each language separately, they do not indicate whether the Rust version improves on the original code or introduces new weaknesses. The delta metric fills this gap by comparing the presence of vulnerabilities before and after transpilation.

Computation Method. For each transpilation iteration, vulnerabilities detected in C/C++ and Rust are compared to determine three outcomes:

- **Removed:** present in C/C++ but absent in Rust,
- **Persisting:** present in both versions,
- **Introduced:** absent in C/C++ but appearing in Rust.

A net security gain is then calculated as:

$$\Delta = \text{Removed}(\mathit{vulnerabilities}) - \text{Introduced}(\mathit{vulnerabilities})$$

Significance. A positive Δ value indicates that transpilation improves overall security, whereas a negative value indicates that new vulnerabilities have been introduced. This metric directly measures whether the tool achieves its fundamental objective: reducing vulnerability exposure.

Weight in the Final Score. The delta metric contributes **15%** to the total security score. This underscores its importance for validating the effectiveness of the transpilation process, while acknowledging its partial dependence on the results of static and dynamic analyses.

4.6 Final Score Analysis

Once all four sub-metrics have been computed and normalized, they are aggregated into a single global security score for each language. This score provides a consolidated, interpretable measure of the overall security posture of the evaluated codebase. Thanks to the normalization process described in Section 4.1, the final scores are directly comparable between languages. Each sub-metric is first mapped to a bounded risk indicator in $[0, 1]$, and the final score is a weighted aggregation of these indicators. In this scale, higher values indicate higher security risk, and lower scores correspond to a stronger security posture. Since the original C/C++ implementation remains unchanged during the entire process, its score is expected to remain constant. In contrast, as the Rust version becomes safer, its score should decrease. A monotonic downward trend in the Rust score, therefore, indicates a security improvement, validating the purpose of the transpilation process.

4.7 Methodological Robustness and Generalization

The proposed methodology is designed to remain robust across variations in tools, configurations, and programming languages. By abstracting vulnerability findings into a shared taxonomy and normalizing them into risk-density metrics, the framework reduces sensitivity to tool-specific artifacts and structural differences in code.

The use of bounded risk indicators further ensures that heterogeneous measurements can be aggregated without introducing scale inconsistencies. Additionally, the methodology is modular: alternative static analyzers, fuzzing engines, or classification approaches can be integrated, provided their outputs can be mapped to the common representation.

Finally, the ML-assisted crash triage component is optional and does not affect the correctness of the evaluation pipeline. This modularity supports extensibility and facilitates adaptation to different evaluation environments and industrial contexts.

5 Results and Observations

This section applies the methodology to assess whether normalization mitigates structural and tooling biases and produces interpretable comparisons.

The following results were obtained using controlled CWE-based implementations and should be interpreted as indicative of methodological behavior rather than as representative of industrial-scale systems.

Experiments were conducted on Ubuntu 24.04.2 LTS using GCC 13.3.0 and Rust 1.94.0-nightly. All binaries were compiled with AFL++ 4.09 (`afl-cc`) and `-O0`, ensuring stable control-flow structure and easier interpretation of sanitizer reports.

To enhance runtime vulnerability detection, programs were compiled with AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan), enabling detection of memory-safety violations and other undefined behaviors relevant to the security evaluation.

5.1 Code Base

A representative and controlled codebase is a prerequisite for the evaluation. As discussed in Section 1, the motivation of this work originates from a transpilation project in which an original C codebase is transformed into an equivalent Rust implementation.

To first evaluate the methodology independently of transpiler bias, we constructed an open-source, rule-based experimental dataset[4] comprising multiple equivalent implementations, each containing specific, well-defined vulnerabilities. For each test case, we implemented one version in C and an equivalent version in Rust. The vulnerabilities were selected from the MITRE Common Weakness Enumeration (CWE) framework [40], ensuring that the experiments are grounded in widely recognized vulnerability definitions.

In this study, we focus on six representative CWEs: CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-121 (Stack-based Buffer Overflow), CWE-127 (Buffer Under-read), CWE-188 (Reliance on Data/Memory Layout), CWE-416 (Use After Free), and CWE-789 (Memory Allocation with Excessive Size Value).

For each CWE, one or more vulnerable code samples were implemented in C and reproduced in Rust with equivalent functionality. The analysis is conducted by grouping all code samples associated with the same CWE and applying the proposed methodology to each group independently. This process yields, for each CWE, a static analysis score, a dynamic analysis score, a memory-related score, and a delta score capturing introduced and removed vulnerabilities. These intermediate results are then aggregated into a final score for each CWE.

5.2 Static Analysis Results

As explained in Section 4, static and dynamic analyses are the main contributors to the global security score, with weights of 30% and 45%, respectively. It is therefore useful to examine them separately before discussing the final aggregated score.

Figure 1 reports the static analysis scores for the code samples, grouped by CWE. Overall, C and Rust obtain similar static scores for most CWEs. This is expected, as static analysis focuses on potential vulnerabilities inferred from code patterns rather than concrete runtime behavior, and is thus less effective at exposing critical memory-safety issues that depend on execution context.

For CWE-121, the static scores for C and Rust differ only moderately. Many memory-related issues detected in C are not fully removed by transpilation and

[4] `https://github.com/defra-forces/c-rust-cwe-dataset`

may still appear in the Rust version, especially under conservative or pattern-based rules. In addition, the normalization process reduces biases due to code size and structural differences, yielding comparable static scores across languages.

By contrast, CWE-119 shows one of the largest gaps between the two implementations. This indicates that, despite normalization, language-level properties can still produce measurable differences in static analysis results, demonstrating that the evaluation methodology captures meaningful variations in vulnerability exposure across languages.

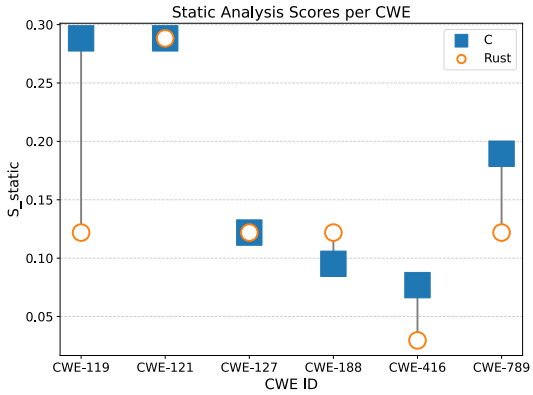

Fig. 1. Static analysis risk scores per CWE for the C (filled square) and Rust (open circle) versions; higher values indicate greater vulnerability exposure. Scores are broadly similar across languages, as static rules flag potential issues from code patterns rather than runtime behavior.

5.3 Dynamic Analysis Results

In contrast to static analysis, Figure 2 highlights a significantly sharper difference between C and Rust in the dynamic analysis results. This difference arises because dynamic analysis executes the program and is therefore more likely to reveal concrete runtime failures, particularly critical memory-safety vulnerabilities. Such vulnerabilities are precisely those that Rust's ownership and borrowing model is designed to prevent.

As a result, C implementations exhibit substantially higher dynamic analysis scores than their Rust counterparts across the evaluated CWEs, indicating lower vulnerability exposure in Rust. These findings reinforce the decision to assign a higher weight (45%) to the dynamic analysis sub-metric, as it captures vulnerabilities that are directly exploitable and have a strong security impact.

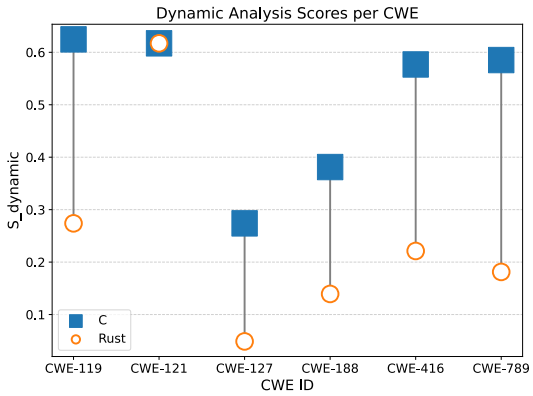

Fig. 2. Dynamic analysis risk scores per CWE (C: filled square, Rust: open circle); higher values indicate greater exposure. Fuzzing reveals a far larger gap than static analysis, with C scoring substantially higher across all CWEs.

5.4 Final Result

The final score aggregates all normalized sub-metric scores, as described in Section 4.6, and provides an objective comparison of the overall security posture of C and Rust for each CWE. It is an aggregated security risk indicator derived from multiple evaluation metrics, where higher values denote greater security risk and **lower values indicate a stronger security posture**.

As shown in Figure 3, C implementations consistently obtain higher final scores than their Rust counterparts across all considered CWEs. Since higher scores correspond to greater risk, this indicates that Rust generally offers a stronger security posture, in line with its stronger memory-safety guarantees. Crucially, the methodology quantifies this difference systematically and reproducibly rather than relying on qualitative judgments.

Beyond the final score, we compute a security indicator (Figure 4) that quantifies the relative improvement in security from migrating C to Rust. Whereas final scores are bounded in $[0, 1]$ and reflect the absolute security risk of each implementation, the security indicator captures the *relative improvement* between the two implementations.

For a given CWE, the indicator is defined as the normalized difference between the final scores of the C and Rust implementations and expressed as a percentage:

$$\text{Security\_Indicator}(\%) = 100 \cdot \frac{S_C - S_{Rust}}{\max(\epsilon, S_C)},$$

where S_C and S_{Rust} denote the bounded final scores of the C and Rust implementations, respectively, and ϵ is a small constant (e.g., 10^{-6}) introduced to avoid division by zero when the baseline score S_C approaches zero.

With this formulation, higher values indicate larger security improvements, values close to zero indicate similar security levels, and negative values indicate a regression, meaning that the Rust implementation exhibits a higher risk score than the original C version.

This indicator can be integrated into a broader evaluation framework alongside metrics assessing correctness and performance, enabling a more comprehensive and interpretable assessment of the impact of transpilation.

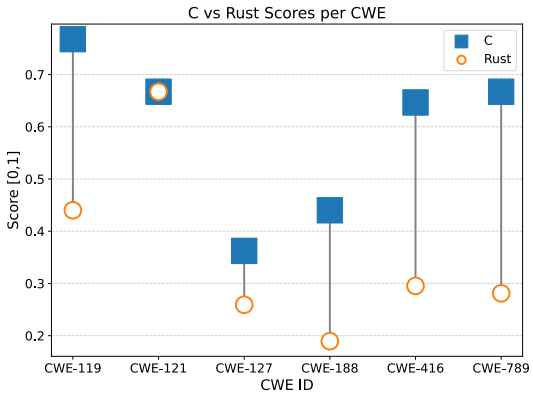

Fig. 3. Aggregated final security score in $[0, 1]$ per CWE (C: filled square, Rust: open circle); higher values denote greater overall risk. C scores above Rust on every CWE except CWE-121, where the two coincide.

6 Conclusion

This paper presents a structured, reproducible methodology for comparing the security posture of functionally equivalent implementations across programming languages, providing an objective, measurable way to assess whether transpilation yields security gains beyond qualitative claims.

The approach combines complementary analytical techniques with a normalization strategy that enables fair cross-language comparisons and quantitative assessment of security differences. Experiments on controlled CWE-based samples show the methodology is feasible and captures meaningful language-driven security differences.

These findings are consistent with the broader shift toward memory-safe languages, such as Google's integration of Rust into Android and reports of fewer memory-related vulnerabilities [19]. Unlike such outcome-focused reports, our contribution is a reusable, language-agnostic methodology for measuring and comparing security gains across languages.

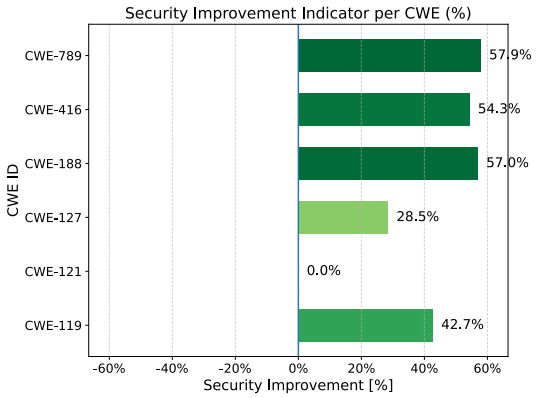

Fig. 4. Per-CWE security improvement indicator: the relative reduction in final score from C to Rust, as a percentage. Positive values denote reduced risk after transpilation, ranging from 0% (CWE-121) to 57.9% (CWE-789).

Several limitations remain. The current evaluation covers only controlled, CWE-based C and Rust implementations, which ensure reproducibility and precise control over vulnerabilities but do not match the complexity and scale of industrial codebases. The methodology also depends on a particular toolchain and configurations; despite mapping tool outputs to a shared vulnerability taxonomy and normalized risk indicators, differences in tools, rule sets, or fuzzing strategies can still influence results, and normalization cannot fully eliminate this variability.

Moreover, the ML-assisted crash triage is evaluated on just 100 samples, limiting statistical confidence and coverage of real-world crash patterns. Finally, calibration parameters and sub-metric weights in the scoring model are design choices that, while grounded in established risk-scoring practices and applied consistently across languages, encode specific risk preferences and may need adaptation to different contexts.

Future work includes evaluating industrial-scale codebases, exploring alternative toolchains and configurations, expanding and diversifying the ML training dataset, and conducting systematic sensitivity analyses of calibration parameters to further strengthen robustness and applicability.

Acknowledgment

This work has been supported by the Royal Higher Institute for Defence (RHID) under the Defence-related Research Action (DEFRA), contract no. 24DEFRA001.

References

1. AFLplusplus Project: AFL++. https://github.com/AFLplusplus/AFLplusplus
2. Aguililla Klein, E.: Security Assessment of Memory Safe/Unsafe Languages: A Framework for Automatic Comparison Between C/C++ and Rust. Master's thesis, Université Libre de Bruxelles (2025)
3. Ahmadi Mehri, V., Arlos, P., Casalicchio, E.: Normalization Framework for Vulnerability Risk Management in Cloud. In: Proceedings of the 8th International Conference on Future Internet of Things and Cloud (FiCloud). IEEE (2021)
4. Al Atiiq, S., Gehrmann, C., Dahlén, K.: Vulnerability Detection in Popular Programming Languages with Language Models. https://arxiv.org/abs/2412.15905 (2024)
5. C2Rust Project: C2Rust: Automated Migration from C to Rust. https://c2rust.com/
6. Cargo-Geiger Project: Cargo-Geiger: Detecting Unsafe Rust Code. https://github.com/geiger-rs/cargo-geiger
7. Croft, R., Xie, Y., Zahedi, M., Babar, M.A., Treude, C.: An Empirical Study of Developers' Discussions about Security Challenges of Different Programming Languages. https://arxiv.org/abs/2107.13723 (2021)
8. De Greef, R., Discepoli, A., Aguililla Klein, E., Engels, T., Hasselmann, K., Paolillo, A.: Towards Macro-Aware C-to-Rust Transpilation (WIP). In: Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2025). pp. 57–61. Association for Computing Machinery (2025). https://doi.org/10.1145/3735452.3735535
9. Defense Advanced Research Projects Agency (DARPA): TRACTOR: Translating All C to Rust. https://www.darpa.mil/research/programs/translating-all-c-to-rust (2024)
10. Dowd, K., Cotter, J.: Exponential Spectral Risk Measures. https://arxiv.org/abs/1103.5409 (2011)
11. Emre, M., Schroeder, R., Dewey, K., Hardekopf, B.: Translating C to safer Rust. Proc. ACM Program. Lang. **5**(OOPSLA) (Oct 2021). https://doi.org/10.1145/3485498
12. Enberg, P., Costa, G.: Introducing Limbo: A Complete Rewrite of SQLite in Rust. https://turso.tech/blog/introducing-limbo-a-complete-rewrite-of-sqlite-in-rust (2024)
13. Engels, T., Discepoli, A., De Greef, R., Aguililla Klein, E., D'Agostino, F., Gunsett, R., Pisane, J., Hasselmann, K., Paolillo, A.: FORCES: An Incremental Transpiler from C/C++ to Rust for Robust and Secure Robotics Systems. In: ICRA 2025 Workshop on Rust for Robotics (R4R). Atlanta, Georgia, USA (May 2025)
14. Farrukh, M., Coskun, B., Palit, T., Polychronakis, M.: SafeTrans: LLM-assisted Transpilation from C to Rust (2025), https://arxiv.org/abs/2505.10708
15. Feng, R., Chen, H., Wang, S., Karim, M.M., Jiang, Q.: LLM-MalDetect: A Large Language Model-Based Method for Android Malware Detection. IEEE Access (2025)
16. Fish Shell Project: Fish 4.0: The Fish of Theseus – Rewriting fish in Rust. https://fishshell.com/blog/rustport/ (2024)
17. Forum of Incident Response and Security Teams (FIRST): Common Vulnerability Scoring System (CVSS). https://www.first.org/cvss/

18. Gasiba, T.E., Amburi, S., Iosif, A.C.: Can Secure Software be Developed in Rust? On Vulnerabilities and Secure Coding Guidelines. International Journal on Advances in Security **17**(1–2), 53–66 (2024), http://www.iariajournals.org/security/
19. Google: Rust in Android: Move Fast, Fix Things. https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html (2025)
20. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: RustBelt: Securing the Foundations of the Rust Programming Language. https://plv.mpi-sws.org/rustbelt/ (2018)
21. Kavian, A., Pourhashem Kallehbasti, M.M., Kazemi, S., Firouzi, E., Ghafari, M.: LLM security guard for code. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. pp. 600–603 (2024)
22. Kharma, M., Choi, S., AlKhanafseh, M., Mohaisen, D.: Security and Quality in LLM-Generated Code: A Multi-Language, Multi-Model Analysis. arXiv preprint arXiv:2502.01853 (2025)
23. Khatry, A., Zhang, R., Pan, J., Wang, Z., Chen, Q., Durrett, G., Dillig, I.: CRUST-Bench: A Comprehensive Benchmark for C-to-Safe-Rust Transpilation. In: Proceedings of the Second Conference on Language Modeling (2025)
24. Koscinski, V., Nelson, M., Okutan, A., Falso, R., Mirakhorli, M.: Conflicting Scores, Confusing Signals: An Empirical Study of Vulnerability Scoring Systems. https://arxiv.org/abs/2508.13644 (2025)
25. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 6 (2018)
26. Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J.R., Hassan, A.E.: In Rust We Trust: A Transpiler from Unsafe C to Safer Rust. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22). pp. 354–355 (2022). https://doi.org/10.1145/3510454.3528640
27. Luo, Y., Zhou, H., Zhang, M., De La Rosa, D., Ahmed, H., Xu, W., Xu, D.: HALURust: Exploiting Hallucinations of Large Language Models to Detect Vulnerabilities in Rust. https://arxiv.org/abs/2503.10793 (2025)
28. National Security Agency: NSA Releases Guidance on How to Protect Against Software Memory Safety Issues. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/ (2022)
29. Ollama, Inc.: Ollama. https://ollama.com (2024)
30. Park, J., Lim, I., Ryu, S.: Battles with false positives in static analysis of JavaScript web applications in the wild. In: Proceedings of the 38th International Conference on Software Engineering Companion. pp. 61–70 (2016)
31. Perkel, J.M.: Why Scientists Are Turning to Rust. Nature **588**(7836), 185–186 (2020). https://doi.org/10.1038/d41586-020-03382-2
32. Rice, H.G.: Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society **74**(2), 358–366 (1953). https://doi.org/10.1090/S0002-9947-1953-0053041-6
33. Rudra Project: Rudra: Finding Memory Safety Bugs in Rust. https://github.com/sslab-gatech/Rudra
34. Rust Fuzz Project: afl.rs. https://github.com/rust-fuzz/afl.rs
35. Rust Project: Clippy: A Collection of Lints to Catch Common Mistakes and Improve Rust Code. https://github.com/rust-lang/rust-clippy
36. Semgrep Inc.: Semgrep. https://semgrep.dev/
37. SonarSource: SonarQube. https://www.sonarsource.com/products/sonarqube/
38. SonarSource: Introducing Support for Rust in SonarQube. https://www.sonarsource.com/blog/introducing-rust-in-sonarqube/ (2025)

39. Stack Overflow Blog: In Rust We Trust: White House Office Urges Memory Safety. https://stackoverflow.blog/2024/12/30/in-rust-we-trust-white-house-office-urges-memory-safety/ (2024)
40. The MITRE Corporation: Common Weakness Enumeration (CWE). https://cwe.mitre.org/
41. Xu, H., Chen, Z., Sun, M., Zhou, Y., Lyu, M.R.: Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs. ACM Trans. Softw. Eng. Methodol. **31**(1) (2021). https://doi.org/10.1145/3466642
42. Zheng, X., Wan, Z., Zhang, Y., Chang, R., Lo, D.: A Closer Look at the Security Risks in the Rust Ecosystem. ACM Trans. Softw. Eng. Methodol. **33**(2) (Dec 2023). https://doi.org/10.1145/3624738