FORCES: An Incremental Transpiler from C/C++ to Rust for Robust and Secure Robotics Systems

Théo Engels[1], Attilio Discepoli[2], Robbe De Greef[2], Esteban Aguililla Klein[3], Francesco D'Agostino[4], Remi Gunsett[4], Jonathan Pisane[4], Ken Hasselmann[1] and Antonio Paolillo[2]

***Abstract*— Unsafe memory accesses are the cause of most cybersecurity vulnerabilities. Robotic systems are not exempt from these risks, especially in defense environments where they are prime targets for cyber threats, and exploiting these vulnerabilities can lead to significant physical consequences.**

To limit the risks associated with memory-unsafe languages in robotic systems, the FOundations for Reliable, CorrEct, and Secure robotic systems (FORCES) project proposes the design of two tools. First, a robust and incremental transpilation tool that enables the conversion of legacy C/C++ code to Rust, thereby enhancing memory safety without sacrificing performance. Second, a comprehensive evaluation framework that establishes metrics for correctness, security, performance, and maintainability to assess the effectiveness of the transpilation process. Both tools will be tested and validated across diverse robotic use cases for the Belgian Defense.

I. Introduction

In early 2024, the White House Office of the National Cyber Director issued a report highlighting the risks associated with languages allowing direct memory manipulation such as C and C++, and urging programmers to adopt memory-safe languages for all new programs, while also applying memory-safe practices to legacy codebases in an effort to prevent vulnerabilities that have led to breaches like the 2014 Heartbleed bug and the 2023 BLASTPASS incident [1]. The European Union, through the Cyber Resilience Act [2] and the NIS2 Directive [3], also recognizes the importance of improving cybersecurity in its member states, highlighting the need for robust security measures in software development and funding initiatives pushing for the adoption of memory-safe programming practices [4]. In March 2024, Google [5] revealed that nearly 70% of the high-criticality vulnerabilities in Android and Chrome stem from memory safety issues. Microsoft [6] has also shown that around 70% of their CVEs are due to memory corruption in C/C++ code. Both companies argue that adopting memory-safe languages like Rust can proactively eliminate these vulnerabilities with minimal performance loss, or even potential performance gains. The US Department of Defense's TRACTOR program aims to automate the translation of C code into Rust in order to prevent and eliminate memory security vulnerabilities in C programs [7]. Collectively, these initiatives underline that memory safety is not only a best practice but a matter of safety and reliability for critical systems, whose failures can generally lead to catastrophic consequences.

[1]Royal Military Academy of Belgium, Brussels, Belgium
[2]Vrije Universiteit Brussel, Brussels, Belgium
[3]Université Libre de Bruxelles, Brussels, Belgium
[4]Thales Belgium, Tubize, Belgium

Robotic systems—such as unmanned aerial, ground and maritime vehicles—are increasingly deployed in high-stakes environments, from surveillance and reconnaissance to demining and payload delivery. The unique complexity and autonomy of these systems, combined with their exposure to cyber threats, amplify the potential impact of security vulnerabilities, especially in defense contexts. The tangible risks and physical consequences associated with compromised robotic systems underscore the urgency of adopting memory-safe practices.

Contributions. We introduce **FORCES**, a toolchain and methodology to support the incremental migration of legacy C/C++ codebases to memory-safe Rust, with a focus on *robotics applications*. FORCES aims to assist the translation of legacy C/C++ cobebases into safe Rust by proposing a novel transpiler (a source-to-source compiler) and a framework for evaluating the transpiled programs through 4 high-level metrics: *correctness*, *performance*, *security* and *code quality*. The transpiler will perform an incremental, function-by-function translation, and each function translation will be evaluated on the basis of these 4 metrics to ensure that the code is correctly translated before being integrated into the resulting codebase.

II. Related Work

A. Legacy Languages and Rust as Safer Alternative

Both academia and industry have documented that C/C++'s manual memory manipulation leads to critical vulnerabilities (e.g., buffer overflows, null-pointer dereferences) [5], [8], [9], [10], motivating a shift towards safer languages. Rust's built-in ownership model and zero-cost abstractions deliver memory safety, even for concurrent code [11], [12], [13], without significantly compromising performance [14], [15], [16], [17]. Its growing adoption in security-sensitive systems [18], [19], its integration into ROS [20], and Rust-translated Linux kernel modules [21], [15], [22] underscore its suitability for defense robotics.

B. Automated Transpilation Efforts

Recent initiatives to automate the migration from C and C++ to Rust showed the feasibility and benefits of transitioning legacy codebases. However, existing solutions fall short on certain points. For instance, Tripuramallu et al. [23] showed that C2Rust [24]—a project funded by DARPA—can only handle C code, converting it to unsafe Rust, and that CRust [25] has limited class transpilation support and cannot handle header files. In the same work, the authors also

proposed a first step to address these shortcomings: a manual transpilation process that maps C/C++ constructs to Rust counterparts to guide automated conversion. Other efforts to improve and build upon these tools have been made: Emre et al. [26] provide an empirical study of unsafety in automatically translated Rust code and introduce techniques leveraging Rust's borrow checker to reduce raw pointer usage; Ling et al. [27] present CRustS2, a fully automated transformation system that refines C2Rust output by systematically reducing unsafe code in function signatures.

C. Evaluation Metrics for Transpiled Code

A critical aspect of adopting a new transpilation approach is the ability to rigorously assess the resulting code across the 4 metrics we identified. State-of-the-art methodologies and tools exist for evaluating these various dimensions.

To assess *correctness*, property-based testing frameworks exist like QuickCheck, Hypothesis, and other Rust-specific tools like Kani, Loom, and Shuttle.

As for *security*, dynamic analysis techniques such as fuzzing [28], instrumentation [29], and dynamic taint analysis [30] are precise but slow. In contrast, static analysis techniques—namely, graph-based [31], distance-based [32], and symbolic execution [33]—offer faster evaluations, although they tend to be less precise. Consequently, these techniques can be used either separately or in a hybrid fashion to effectively identify bugs and vulnerabilities.

Benchmarking tools (e.g., benchkit [34] or Google Benchmark) that precisely measure code *performance*, along with profiling tools (e.g., gprof or perf) that identify performance bottlenecks, can pinpoint the parts of the code that most affect system performance, thereby guiding iterative optimization efforts.

Static analysis tools like Clang Static Analyzer and Clippy help assess *code quality*. Moreover, platforms such as SonarQube and CodeClimate provide more comprehensive assessments of code maintainability, including cyclomatic complexity, duplication, and technical debt.

III. Challenges Identified

Existing transpilation tools like C2Rust [24], CRust [25], and Corrode [35] produce non-idiomatic, C-like code that is difficult to maintain. We identified several challenges during the analysis of these tools, including (i) the transpilation of preprocessor directives (such as header inclusion, macros, and conditional compilation), (ii) the translation of object-oriented code (which can be aided by tools such as (auto)cxx [36], [37]), (iii) concurrency management (which is not standard in C), (iv) the translation of errors and exceptions into Rust's `Result<T>` and `panic!()` mechanisms, and (v) the generation of idiomatic Rust code that leverages the standard library and functional programming features.

Another significant challenge is the current lack of evaluation tools to verify that the transpilation has been executed correctly. Such tools would allow for a thorough assessment of the correctness, security, performance, and overall quality of the produced source code. Additionally, they should support hybrid binaries, enabling the evaluation of code that has been only partially transpiled.

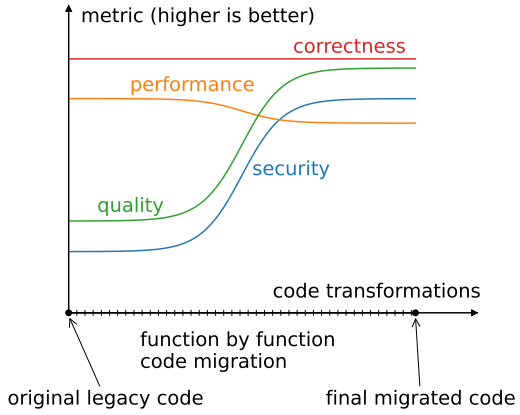

Fig. 1. Impact of fine-grained migration on performance, security, correctness, and code quality. The horizontal axis shows the percentage of source code transpiled to Rust; the vertical axis shows metric scores.

IV. FORCES Objectives

A. Automated Fine-Grained Code Transpilation from C/C++ to Rust

The main objective of the project is to develop an automated system to convert C/C++ code to memory-safe Rust. This transpilation tool will operate at a function-by-function level, offering granular control to allow end users to transpile only part of their application (e.g., the safety-critical part) or the whole codebase (see Figure 1). The transpiler aims to convert as much of the legacy codebase as possible while reducing potential memory-related vulnerabilities without affecting too much the performance of the original code. The validation of the transpiler will focus on robotics software patterns (e.g., ROS nodes, kernel device drivers, etc.) and will provide guidance through manual transpilation for sections that cannot be automatically transpiled.

To achieve this, the transpiler draws substantial inspiration from established compiler architectures and comprises three principal components: a frontend, middleware, and a backend. Unlike traditional compilers, this transpiler operates on a source-to-source basis. Consequently, the middleware's intermediate representation (IR) is an abstract syntax tree (AST) as opposed to a lower-level assembly-like language such as LLVM IR. This intermediate AST takes inspiration from the Rust AST, since Rust is the intended target for the backend. However, the AST is designed to be sufficiently distinct from Rust to allow for a smooth conversion process between the two languages.

Within each of the three stages, the transpiler performs a series of incremental passes. Each pass serves to refine or alter aspects of the IR. Some passes are essential for achieving accurate transpilation while others are optional, with the aim of enhancing the readability or maintainability of the code. Furthermore, these incremental passes simplify the design and debugging of the transpilation process by iso-

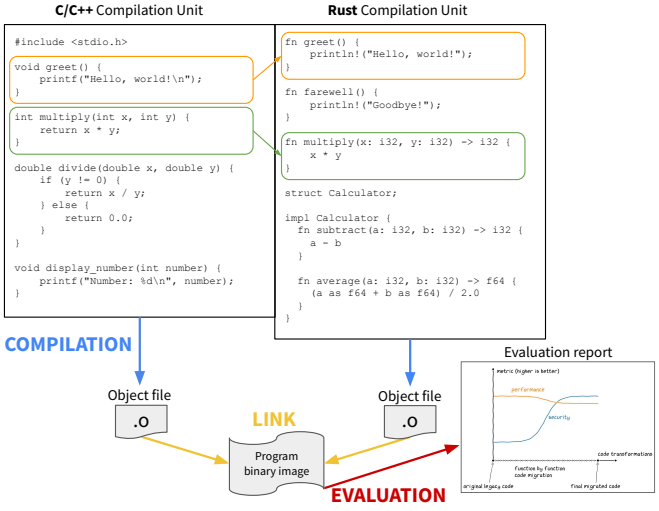

Fig. 2. Illustration of the transpilation, compilation and linking process. C/C++ source is partially transpiled to Rust; both the untouched C/C++ and the generated Rust code are then compiled and linked into a single binary

lating concerns. This mechanism also permits the activation or deactivation of specific features by choosing whether to execute a particular pass.

Figure 2 illustrates the process of the function-by-function transpilation workflow. For each C/C++ compilation unit, an initially empty corresponding Rust compilation unit is created. The tool then selects a function from the C/C++ unit, transpiles it into the Rust unit, and the resulting set of compilation units is compiled and linked to produce a new binary program. This binary is subsequently evaluated against the established metrics, generating a new evaluation point on the x-axis of Figure 1. This iterative process enables the progressive improvement of code safety and performance. Various heuristics for function selection—such as prioritizing functions based on their size or execution time footprint in test cases—are explored to identify those that most significantly affect overall system security, safety and efficiency.

B. Metrics and Evaluation Methodologies

Despite the availability of multiple tools to measure security, correctness, performance, and quality of the code, an integrated approach that combines the assessments of all four metrics in the context of defense cyber-physical systems is missing. The FORCES project aims to fill this gap by developing a comprehensive evaluation framework that systematically captures these key performance indicators.

Furthermore, those metrics allow us to precisely quantify the impact of our transpilation tool and find weak spots that require further processing. Since some amount of unsafe Rust is unavoidable for low-level software, it is essential to demonstrate that this unsafety is reduced—or at the very least, not greater—than that present in the original C/C++ code. Moreover, as C/C++ continues to dominate in critical systems, ensuring seamless interoperability with Rust is crucial. Therefore, in the context of our incremental and hybrid transpilation approach, we must also assess the impact of using the foreign function interface on Rust's overall security guarantees.

C. Defense use cases

The third objective of FORCES focuses on defining and validating real-world defense robotics applications as test cases across a diverse range of defense use cases derived from both past and ongoing initiatives at the Royal Military Academy of Belgium, covering air, ground, and maritime robotics. Controlled robotic testbeds will be used to replicate experimental environments, thereby facilitating the application of the transpilation methodology and evaluation metrics developed in earlier phases.

One concrete use case involves Universal Robots arms that are used in demining projects. We discovered that the API for these cobots (`ur_rtde` v1.6.0) presents one buffer-overflow vulnerability that causes shaky joint motions, a significant risk when handling explosives. Transpiling this codebase would eliminate such vulnerabilities, making the use of the arm safer.

Over the four-year project, the scope of these use cases will be progressively expanded to align with evolving defense requirements, incorporate operational feedback, and encompass increasingly complex applications that demand a more mature transpiler. A dedicated field test showcasing a transpiled codebase will be organized to demonstrate and validate the effectiveness of the transpilation process.

V. Project Timeline and Milestones

The FORCES project, which officially started in December 2024, is structured around multiple milestones as illustrated in Figure 3. Future publications will document project progress, providing concrete transpilation examples and quantitative benchmarks.

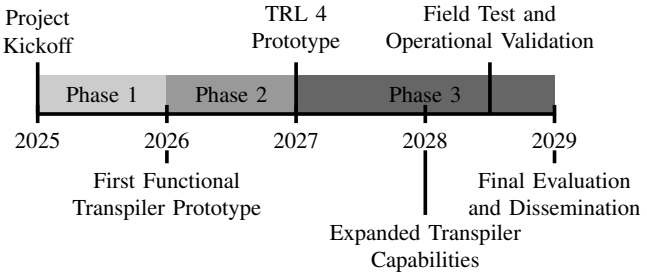

Fig. 3. FORCES project timeline highlighting major milestones divided into three phases. **Phase 1** (Foundational Research, 2025–2026) focuses on developing the transpilation methodology, defining evaluation metrics, and selecting relevant defense use cases. **Phase 2** (Prototype Development, 2026–2027) aims to build and integrate a TRL 4 transpiler prototype into robotic testbeds while refining the evaluation pipeline. **Phase 3** (Operational Enhancement, 2027–2029) involves iterative tool improvement to achieve TRL 5, culminating in comprehensive field trials and the final release of the complete toolchain.

VI. Conclusions

FORCES presents an approach for enhancing the security of legacy C/C++ code through an incremental transpilation

to Rust. By performing a fine-grained, function-by-function transpilation using a comprehensive evaluation framework, the FORCES project aims to improve memory security and to contribute to the foundations of the next generation of robotic systems, built on memory-safe, reliable software, thereby enhancing their resilience against cyber threats.

ACKNOWLEDGMENT

This work is supported by the Belgian Ministry of Defence under the Defence-related Research Action (DEFRA), contract number 24DEFRA001, within the FORCES project.

REFERENCES

[1] "Press Release: Future Software Should Be Memory Safe," Feb. 2024. [Online]. Available: https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

[2] "Cyber Resilience Act | Shaping Europe's digital future," https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act.

[3] "NIS2 Directive: New rules on cybersecurity of network and information systems | Shaping Europe's digital future," https://digital-strategy.ec.europa.eu/en/policies/nis2-directive, Jan. 2025.

[4] "Logical Foundations for the Future of Safe Systems Programming | RustBelt Project | Fact Sheet | H2020," https://cordis.europa.eu/project/id/683289.

[5] "Secure by Design: Google's Perspective on Memory Safety," https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/.

[6] "We need a safer systems programming language | MSRC Blog | Microsoft Security Response Center," https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/.

[7] D. O. DEFENSE, "Translating All C TO Rust (TRACTOR)," 2024. [Online]. Available: https://sam.gov/opp/1e45d648886b4e9ca91890285af77eb7/view

[8] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against C and C++ programs," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 17:1–17:28, Jun. 2012. [Online]. Available: https://dl.acm.org/doi/10.1145/2187671.2187679

[9] "Modern C++ Won't Save Us · Alex Gaynor," https://alexgaynor.net/2019/apr/21/modern-c++-wont-save-us/.

[10] I. S. R. Group, "What is memory safety and why does it matter?" https://www.memorysafety.org/docs/memory-safety/.

[11] "Safe Systems Programming in Rust – Communications of the ACM," Apr. 2021.

[12] "Rust Vulnerability Analysis and Maturity Challenges," https://insights.sei.cmu.edu/blog/rust-vulnerability-analysis-and-maturity-challenges/, Jan. 2023.

[13] T. H. News, "Google's Shift to Rust Programming Cuts Android Memory Vulnerabilities by 68%," https://thehackernews.com/2024/09/googles-shift-to-rust-programming-cuts.html.

[14] Y. Zhang, Y. Zhang, G. Portokalidis, and J. Xu, "Towards Understanding the Runtime Performance of Rust," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, Jan. 2023, pp. 1–6. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3559494

[15] I. S. R. Group, "Rust in the Linux Kernel: Just the Beginning," https://www.memorysafety.org/blog/rust-in-linux-just-the-beginning/, Oct. 2022.

[16] F. Suchert and J. Castrillon, "Stamp-rust: Language and performance comparison to c on transactional benchmarks," in *Benchmarking, Measuring, and Optimizing*, A. Gainaru, C. Zhang, and C. Luo, Eds. Cham: Springer International Publishing, 2023, pp. 160–175.

[17] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamarić, and L. Ryzhyk, "System programming in rust: Beyond safety," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 156–161. [Online]. Available: https://doi.org/10.1145/3102980.3103006

[18] Internet Security Research Group, "Rustls," https://www.memorysafety.org/initiative/rustls/.

[19] A. Sharma, S. Sharma, S. Torres-Arias, and A. Machiry, "Rust for embedded systems: Current state, challenges and open problems (extended report)," 2024. [Online]. Available: https://arxiv.org/abs/2311.05063

[20] "Rust is for Robotics," https://robotics.rs/.

[21] S.-F. Chen and Y.-S. Wu, "Linux kernel module development with rust," in *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, 2022, pp. 1–2.

[22] "The First Rust-Written Network PHY Driver Set To Land In Linux 6.8," https://www.phoronix.com/news/Linux-6.8-Rust-PHY-Driver.

[23] D. Tripuramallu, S. Singh, S. Deshmukh, S. Pinisetty, S. A. Shivaji, R. Balusamy, and A. Bandeppa, "Towards a transpiler for c/c++ to safer rust," 2024. [Online]. Available: https://arxiv.org/abs/2401.08264

[24] "immunant/c2rust," Apr. 2025. [Online]. Available: https://github.com/immunant/c2rust

[25] N. Shetty, N. Saldanha, and M. N. Thippeswamy, "Crust: A c/c++ to rust transpiler using a "nano-parser methodology" to avoid c/c++ safety issues in legacy code," in *Emerging Research in Computing, Information, Communication and Applications*, N. R. Shetty, L. M. Patnaik, H. C. Nagaraj, P. N. Hamsavath, and N. Nalini, Eds. Singapore: Springer Singapore, 2019, pp. 241–250.

[26] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating C to safer Rust," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021. [Online]. Available: https://doi.org/10.1145/3485498

[27] M. Ling, Y. Yu, H. Wu, Y. Wang, J. R. Cordy, and A. E. Hassan, "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*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 354–355. [Online]. Available: https://dl.acm.org/doi/10.1145/3510454.3528640

[28] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[29] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, no. 6, pp. 89–100, Jun. 2007. [Online]. Available: https://doi.org/10.1145/1273442.1250746

[30] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 317–331, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/5504796

[31] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code," in *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/discovre-efficient-cross-architecture-identification-bugs-binary-code.pdf

[32] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: a semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 896–899. [Online]. Available: https://doi.org/10.1145/3238147.3240480

[33] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: https://dl.acm.org/doi/10.1145/360248.360252

[34] "open-s4c/benchkit," Apr. 2025. [Online]. Available: https://github.com/open-s4c/benchkit

[35] J. Sharp, "jameysharp/corrode," Mar. 2025. [Online]. Available: https://github.com/jameysharp/corrode

[36] "google/autocxx," Apr. 2025. [Online]. Available: https://github.com/google/autocxx

[37] D. Tolnay, "dtolnay/cxx," Apr. 2025. [Online]. Available: https://github.com/dtolnay/cxx