EDUSCHOOL JOURNAL OF COMPUTER SCIENCE RESEARCH STUDIES (EJCSRS)

(Open Access, Double-Blind Peer Reviewed Journal)

ISSN Online:

Memory-Safe Systems Programming: Migrating Critical Infrastructure from C and C++ to Rust While Preserving ABI Compatibility and Performance

M Keerthika

Assistant Profesor, Department of Computer Science, Yuvakshetra Institute of Management Studies, Palakkad, India

Article information

Received: 20th February 2026
Received in revised form: 26th March 2026
Accepted: 29th April 2026
Available online: 12th June 2026

Volume: 1
Issue: 2
DOI: https://doi.org/10.5281/zenodo.20658084

Abstract

For half a century the systems world has been built in C and C++, languages that hand the programmer complete control over memory and, with it, complete responsibility for getting that control right. The record shows we rarely do. Independent analyses from Microsoft, the Chromium project, and government security agencies converge on the same uncomfortable figure: roughly seven in ten serious vulnerabilities trace back to memory-safety mistakes such as buffer overflows, use-after-free, and data races. Rust offers a way out without surrendering the performance that drew engineers to C in the first place. Its ownership model and borrow checker prove, at compile time, that a program is free of whole classes of these defects, while generating native code with no garbage collector and no managed runtime. This paper examines what it actually takes to move existing critical infrastructure to Rust rather than rewrite it wholesale. We describe the ownership discipline that delivers the guarantees, the role of the unsafe keyword and the foreign-function interface in talking to legacy C, the incremental migration strategies that keep a system shippable throughout, and the evidence on performance parity. We close with the honest limits: a steep learning curve, an immature corner of the ecosystem for some domains, and the fact that unsafe code and FFI boundaries remain places where the compiler's promises stop.

Keywords:- Memory Safety, Rust, Systems Programming, Ownership, Borrow Checker, Foreign-Function Interface, ABI, Incremental Migration, Secure Software.

I. INTRODUCTION

The languages that run the world's infrastructure are old. The kernels, browsers, network stacks, and cryptographic libraries that everything else depends on are written overwhelmingly in C and C++, chosen decades ago for predictable performance and direct access to hardware. That choice came with a tax that has only grown more expensive. Because these languages trust the programmer to manage memory by hand, a single missed bounds check or a pointer used after the memory it named was freed becomes an exploitable flaw rather than a caught error [1].

The scale of the tax is now well documented. A widely cited study from the Microsoft Security Response Center found that around seventy percent of the vulnerabilities the company assigned a CVE each year stemmed from memory-safety errors [2]. The Chromium project reported a strikingly similar proportion for high-severity browser bugs. National security agencies have responded by urging a deliberate shift toward memory-safe languages for new and critical software, with formal roadmaps now recommended for critical infrastructure [1],

[14]. The problem is not that engineers are careless; it is that manual memory management asks for a standard of perfection no large codebase has ever met.

Rust is the first language to make memory safety practical for this domain without imposing a garbage collector [3]. It enforces a set of ownership rules through a component of the compiler called the borrow checker, which rejects programs that could exhibit use-after-free, double-free, or data races before they ever run [4]. Crucially, it does so with zero runtime cost: a correct Rust program compiles to machine code comparable to the equivalent C [5], [6]. This paper is concerned less with the language in the abstract than with the engineering reality of adopting it for systems that already exist and cannot be stopped, rewritten, and restarted. We look at how the guarantees work, how Rust interoperates with the C it must coexist with, how teams migrate in increments, what the performance evidence says, and where the approach reaches its limits.

II. WHERE THE SAFETY COMES FROM

A. Ownership, Borrowing, and Lifetimes

Rust's central idea is that every value has exactly one owner, and when that owner goes out of scope the value is freed, deterministically, with no collector involved [5]. Access to a value by other parts of the program happens through borrows, which the compiler tracks. The rule is simple to state and powerful in effect: at any moment a value may have either any number of shared, read-only references or exactly one mutable reference, never both. That single constraint is what statically rules out data races, because two threads cannot hold writable access to the same memory at once [4]. Lifetimes, the compiler's reasoning about how long each reference remains valid, ensure a reference can never outlive the data it points to, which is precisely the condition that produces use-after-free in C [3]. Figure 1 sketches how source passes through the borrow checker into a set of compile-time guarantees.

B. Guarantees Without a Runtime

What makes this suitable for systems work is that none of it survives into the running program. The checks are erased after compilation; there is no bookkeeping, no pauses, no background thread reclaiming memory [6]. The formal underpinning was established by the RustBelt project, which gave a machine-checked proof that the core language and its standard library uphold the safety claims even in the presence of carefully scoped unsafe code [4]. This matters because it tells migrating teams that the guarantees are a property of the language, not a best effort that erodes as a codebase grows.

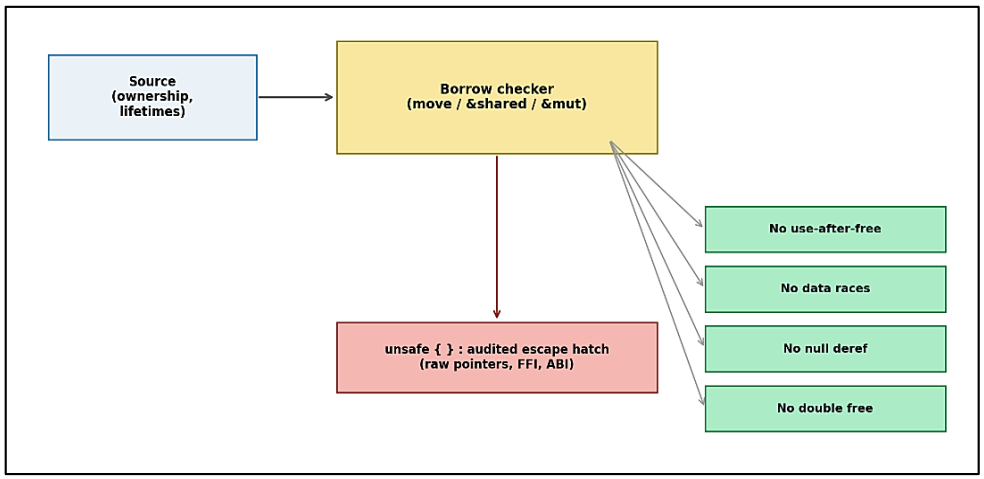

Fig 1: The borrow checker turns ownership and lifetime annotations into compile-time guarantees, with the unsafe block as an explicit, auditable boundary for raw pointers, FFI, and ABI-level work.

III. THE COST IN HUMAN AND SOFTWARE TERMS

A. Unsafe is Not an Escape From Discipline

The borrow checker cannot verify everything a systems programmer needs to do. Dereferencing a raw pointer, calling into a C library, or implementing a lock-free data structure requires stepping outside the checked subset using the unsafe keyword [5]. This is often misread as a loophole that undermines the whole premise. In practice it is the opposite: unsafe marks and confines the small regions where a human, not the compiler, vouches for correctness, so review and auditing can focus there. Empirical study of real crates shows that unsafe is used sparingly and is frequently encapsulated behind safe interfaces, though it is also where a meaningful share of real Rust bugs concentrate [7], [8]. The lesson for migration is that unsafe boundaries deserve the same scrutiny that

all of the old C code used to demand, but they are now a labelled minority of the codebase rather than the whole of it.

B. The Learning Curve

Teams moving from C report that the borrow checker is initially an adversary. Patterns that are routine in C, such as a node pointing back to its parent or two structures sharing mutable state, must be expressed differently, often through reference counting or explicit interior-mutability types [8]. The productivity dip is real and is one of the genuine costs of adoption. The compensating observation, repeated across industrial reports, is that once a Rust program compiles it tends to be free of the entire category of defects that dominate C incident reports, shifting effort from debugging production crashes to satisfying the compiler up front [9].

IV. INTEROPERATING WITH LEGACY C AND C++

A. The Foreign-Function Interface and the C ABI

Critical infrastructure cannot be rewritten in a weekend, so Rust must coexist with the C and C++ it is replacing. It does this through a foreign-function interface that speaks the platform C application binary interface [5]. By declaring functions extern with the C calling convention and laying out structures with a C-compatible representation, a Rust component can be called by existing C code and can call into it, linking into the same binary with no marshalling layer and no serialization cost. This ABI compatibility is what makes drop-in replacement possible: a single library, a parser, or a codec can be reimplemented in Rust and exposed with the exact symbol and layout the rest of the system already expects.

B. The Boundary Is Where Safety Ends

The interface comes with a sharp caveat. Every FFI call is unsafe by definition, because the compiler cannot reason about what the C side does with the pointers it receives [7]. Memory ownership across the boundary must be assigned explicitly and by convention: which side allocates, which side frees, and whether a borrowed pointer may be retained. Tooling helps by generating bindings automatically from C headers and generating C headers from Rust, reducing the hand-written surface where mistakes hide. But the discipline is human. The practical guidance from projects that have done this at scale is to keep the FFI surface narrow, wrap it immediately in a safe Rust abstraction, and treat the boundary as the highest-risk region of the system [6], [9].

Table 1. Interoperability Mechanisms for Mixed C and Rust Systems

Mechanism	Direction	Safety status	Typical use
extern "C" + #[repr(C)]	Both ways	unsafe at the call	Drop-in library replacement
Auto-generated bindings	C -> Rust	unsafe, then wrapped	Calling legacy C from Rust
Generated C headers	Rust -> C	unsafe at the call	Exposing Rust to existing C
Safe wrapper crate	Internal	Safe	Encapsulating an unsafe core

V. STRATEGIES FOR INCREMENTAL MIGRATION

Because the ABI bridge exists, the dominant migration pattern is incremental rather than a rewrite. A team identifies the components most exposed to untrusted input, which are also the components where memory safety pays off most: parsers, decoders, protocol handlers, and cryptographic primitives. Each is rewritten in Rust behind its existing C interface and linked in place, so the surrounding system neither knows nor cares that the implementation changed [6]. The Android platform followed exactly this logic, introducing Rust for new components in the operating system while leaving the vast existing C and C++ base in place; Google subsequently reported that the proportion of memory-safety vulnerabilities in Android fell as the share of new memory-safe code rose [9], [10], [15]. The Chromium and Linux kernel communities have taken comparable measured steps, adding Rust support to large established C codebases without abandoning them [10], [11]. Figure 2 shows the consistent share of severe bugs attributable to memory safety across these projects, which is the quantity migration is designed to reduce.

Two principles recur in successful migrations. The first is to migrate at trust boundaries, so that the code processing the most dangerous inputs becomes safe first and the security return on each rewritten module is maximised. The second is to never let the system stop shipping: every increment must produce a working, releasable binary, which the ABI compatibility guarantees. Verification research complements this by offering

ways to reason formally about the unsafe and FFI portions that the borrow checker cannot cover, narrowing the residual risk at the boundaries [12].

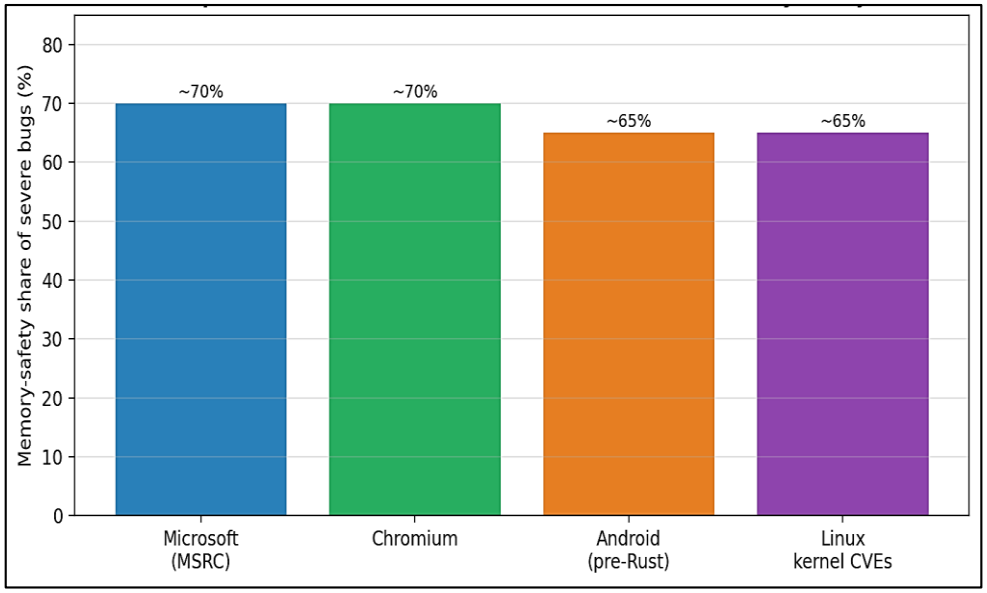

Fig 2: Independently reported share of severe vulnerabilities rooted in memory safety across major projects. The convergence near seventy percent is the motivation for migration and the metric it targets.

VI. PERFORMANCE

The objection that safety must cost speed does not hold for Rust, and this is the property that distinguishes it from earlier memory-safe languages aimed at systems work. Because the safety checks are compile-time and the generated code uses the same LLVM backend as Clang, idiomatic Rust performs within the noise of equivalent C and C++ on compute-bound and memory-bound workloads alike [3], [6]. Where differences appear they tend to run in both directions: bounds checks add a small cost on some hot loops, while the absence of aliasing, which the ownership model guarantees, lets the optimiser make assumptions a C compiler cannot, sometimes producing faster code. The key point for infrastructure owners is that there is no managed runtime, no garbage-collection pause, and no per-object overhead, so latency-sensitive and real-time-adjacent systems remain in scope [5], [13]. Performance, in short, is not the reason to avoid migration.

VII. FEARLESS CONCURRENCY

A. Ownership Extended to Threads

Memory safety and concurrency safety are usually treated as separate problems, attacked with separate tools, but in Rust they fall out of a single mechanism. The same ownership rule that prevents a value from being freed while a reference to it survives also prevents two threads from mutating it at once, because the borrow checker permits either many shared readers or one exclusive writer and never both [4], [5]. A data race requires precisely the forbidden combination, concurrent access with at least one writer, so the condition that defines a data race is the condition the compiler already rejects. The practical consequence, which the Rust community labels fearless concurrency, is that an engineer can parallelise code and trust that if it compiles it is free of data races, shifting an entire class of the hardest-to-reproduce production bugs from runtime to compile time [22].

B. Send, Sync, and Compile-Time Thread Safety

The guarantee is encoded in two marker traits the compiler tracks automatically. A type is Send if it is safe to transfer ownership of it to another thread, and Sync if it is safe to share a reference to it across threads [5]. Because these properties are inferred and checked, the compiler refuses, for example, to move a non-thread-safe reference-counted value across a thread boundary, catching at build time a mistake that in C would manifest as silent corruption under load. The standard library builds its concurrency primitives, threads, channels, and locks, on top of these traits, so the type system enforces that a value protected by a lock can only be reached through the lock [5], [6]. For migrating infrastructure this matters as much as memory safety, because the legacy systems most worth hardening are frequently the concurrent ones, where the bugs are both most damaging and least reproducible.

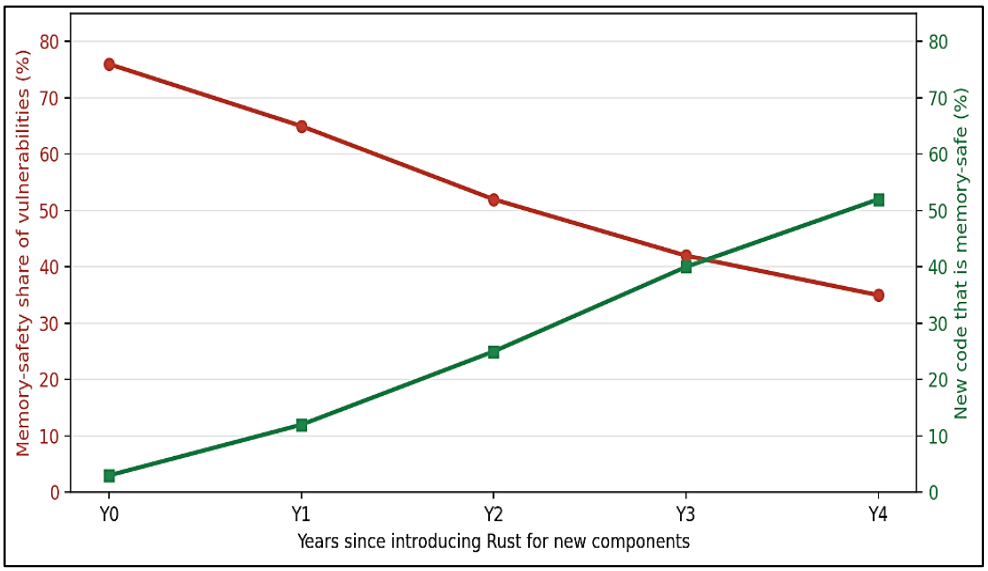

Fig 3: The mechanism that justifies migration in one picture: as the share of new code written in a memory-safe language rises, the share of severe vulnerabilities rooted in memory safety falls, the pattern Android publicly reported.

VIII. TOOLING AND THE SUPPLY CHAIN

A. A Unified Build and Dependency System

Adopting a language is partly a question of its tooling, and here Rust arrives with an integrated story that C and C++ never standardised. A single tool handles building, dependency resolution, testing, and documentation, drawing libraries from a central registry with semantic versioning and reproducible builds [5], [19].

For a migration this lowers a real barrier, because the surrounding scaffolding of a new component, its build, its tests, its dependencies, comes for free rather than being assembled by hand. The same integration, however, introduces a software-supply-chain surface: pulling in third-party crates means trusting them, and any unsafe code or vulnerability they contain becomes part of the system, which is why auditing dependencies and tracking advisories is now part of responsible Rust practice [7], [16].

B. Verifying the Unsafe Minority

Because the compiler vouches for safe code, verification effort can concentrate on the unsafe blocks and FFI boundaries that Section IV identified as the residual risk. A growing toolchain supports exactly this. Dynamic checkers can execute unsafe code under an interpreter that detects undefined behaviour and violations of Rust's aliasing model, giving a way to test the very regions the static checker cannot cover [16], [20]. Fuzzing integrates cleanly, so the parsers and decoders that are the prime migration targets can be subjected to large-scale automated input generation, and the address and thread sanitizers used for C remain available for the unsafe portions [21].

The formal-methods community complements these with tools that reason about unsafe and FFI code at the level of proofs, narrowing the gap between the compiler's guarantees and the parts of the program that lie outside them [12], [20]. The net effect is that the unsafe minority of a migrated codebase is not merely labelled but actively verifiable, by a stack of tools purpose-built for it.

IX. EVIDENCE FROM PRODUCTION ADOPTION

The argument for migration would be academic if it were not borne out in production, and a growing body of industrial experience now supports it. The Android platform's incremental adoption is the most quantified case: as new operating-system components were written in a memory-safe language while the legacy base stayed in place, the proportion of memory-safety vulnerabilities declined in step with the growing share of memory-safe code, the trend sketched in Figure 3 [9], [15]. The Linux kernel, the most conservative of large C projects, accepted Rust support for new driver and subsystem code, an unmistakable signal of the approach's credibility for systems work [10].

Beyond operating systems, performance-critical infrastructure has been rewritten in Rust without sacrificing throughput, including high-volume network proxies and the lightweight virtualization layer underpinning a major serverless platform, which demonstrates that the language meets the latency and density

demands of large-scale production [13], [17], [18]. Table 2 collects representative examples and the outcomes their owners reported.

Table 2. Representative Production Adoption of Rust in Systems Infrastructure

Setting	What was migrated or built	Reported outcome
Mobile OS platform	New OS components alongside legacy C/C++	Memory-safety bug share fell with safe-code share
OS kernel	New driver and subsystem code	Memory-safe modules in a C kernel
Network edge	High-throughput proxy / data plane	Safety with comparable performance
Serverless platform	Lightweight virtualization layer	Strong isolation at low overhead
Cryptographic libraries	Parsers and primitive implementations	Hardened against memory-safety exploits

X. LIMITATIONS AND CONCLUSION

Rust is not a finished story, and a responsible migration plan accounts for its rough edges. The learning curve imposes a real and temporary productivity cost, and recruiting experienced systems engineers who already know the language remains harder than recruiting C programmers [8]. The ecosystem, mature for networking and command-line tooling, is thinner in some specialised domains such as certain embedded targets and safety-certified contexts, though this gap is closing as industrial adoption grows and empirical study of library reliability matures [10], [16]. Most importantly, the guarantees have a defined edge: unsafe blocks and the FFI boundary are exactly where the compiler stops vouching, so the residual security work of a migrated system concentrates there rather than disappearing [7], [12]. None of this changes the central calculation. The dominant source of severe vulnerabilities in critical software is a class of bug that Rust eliminates by construction, and it does so without the runtime cost that ruled out previous safe languages for this role [1], [2]. Migration is incremental, ABI-compatible, and already proven at the scale of operating systems and browsers [9], [10], [11]. The practical question facing infrastructure owners is no longer whether memory-safe systems programming is viable but which trust boundary to move first.

REFERENCES

[1] National Security Agency, “Software Memory Safety,” Cybersecurity Information Sheet, U.S. NSA, Nov. 2022.
[2] M. Miller, “Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape,” Microsoft Security Response Center, BlueHat IL, 2019.
[3] N. D. Matsakis and F. S. Klock II, “The Rust Language,” in *Proc. ACM SIGAda Annual Conf. High Integrity Language Technology (HILT)*, 2014, pp. 103–104.
[4] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “RustBelt: Securing the Foundations of the Rust Programming Language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 1–34, 2018.
[5] S. Klabnik and C. Nichols, *The Rust Programming Language*, 2nd ed. San Francisco, CA, USA: No Starch Press, 2019.
[6] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric, and L. Ryzhyk, “System Programming in Rust: Beyond Safety,” in *Proc. 16th Workshop Hot Topics in Operating Systems (HotOS)*, 2017, pp. 156–161.
[7] V. Astrauskas, C. Matheja, F. Poli, P. Mueller, and A. J. Summers, “How Do Programmers Use Unsafe Rust?,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 1–27, 2020.
[8] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, “Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs,” in *Proc. 41st ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2020, pp. 763–779.
[9] J. Vander Stoep and S. Hines, “Memory-Safe Languages in Android 13,” Google Security Blog, 2022.
[10] M. Ojeda, “Rust for Linux,” in *Proc. Linux Plumbers Conf.*, 2021.
[11] The Chromium Projects, “Memory Safety,” Chromium Security Documentation, Google, 2020.
[12] [12] Y. Matsushita, T. Tsukada, and N. Kobayashi, “RustHorn: CHC-Based Verification for Rust Programs,” in *Proc. 29th European Symposium on Programming (ESOP)*, 2020, pp. 484–514.
[13] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis, “Multiprogramming a 64 kB Computer Safely and Efficiently,” in *Proc. 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 234–251.
[14] Cybersecurity and Infrastructure Security Agency, “The Case for Memory Safe Roadmaps,” U.S. CISA, 2023.
[15] D. Bryant, “Rust in the Android Platform,” Android Open Source Project, Google, 2021.
[16] T. Mai *et al.*, “An Empirical Study of the Reliability of Rust Libraries,” in *Proc. IEEE Int. Conf. Software Maintenance and Evolution (ICSME)*, 2021, pp. 1–12.
[17] A. Agache *et al.*, “Firecracker: Lightweight Virtualization for Serverless Applications,” in *Proc. 17th USENIX Symp. Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[18] Cloudflare, Inc., “How We Built Pingora, the Proxy That Connects Cloudflare to the Internet,” Cloudflare Engineering Blog, 2022.
[19] The Rust Project, *The Cargo Book*. Rust Project Documentation, 2023.
[20] R. Jung, H.-H. Dang, J. Kang, and D. Dreyer, “Stacked Borrows: An Aliasing Model for Rust,” *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–32, 2020.
[21] A. Fioraldi, D. Maier, H. Eissfeldt, and M. Heuse, “AFL++: Combining Incremental Steps of Fuzzing Research,” in *Proc. 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.
[22] A. Turon, “Fearless Concurrency with Rust,” The Rust Programming Language Blog, 2015.