An Empirical Analysis of Rust Integration in Android Open Source Project

Yinte Fan*
Huazhong University of Science and Technology
Wuhan, China
m202572248@hust.edu.cn

Chao Wang*
Huazhong University of Science and Technology
Wuhan, China
chaowang_@hust.edu.cn

Zikan Dong†
Huazhong University of Science and Technology
Wuhan, China
zikandong@hust.edu.cn

Tianming Liu
Huazhong University of Science and Technology
Wuhan, China
tmliu@hust.edu.cn

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

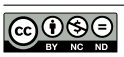
The Android Open Source Project (AOSP) powers the world's largest mobile ecosystem, yet memory-safety vulnerabilities remain prevalent in native components due to the inherent risks of languages like C and C++, which allow manual memory management and unsafe operations. To mitigate these risks, AOSP began adopting Rust, a language with compile-time memory-safety guarantees, in 2021. However, the trajectory, mechanisms, and effectiveness of this transition remain empirically uncharted. We present the first longitudinal and systematic empirical study of Rust in AOSP. By analyzing 16 quarterly snapshots from January 2021 to October 2024, we track adoption trends and map the distribution of Rust code across 67 components. We further align Android Security Bulletin vulnerabilities with these components to assess memory-safety outcomes before and after Rust adoption, while also identifying the remaining non-Rust hotspots that may warrant future migration. In addition, we characterize platform-level integration mechanisms and examine unsafe governance through a large corpus of SAFETY comments, from which we distill actionable best practices and anti-patterns. Taken together, these results clarify Rust's role in AOSP by shedding light on its adoption, integration, and governance in a safety-critical mobile platform. They also provide industry practitioners with an empirical foundation and practical guidance for similar memory-safe migrations in large-scale production systems.

CCS Concepts

• **Software and its engineering → Software evolution**.

*Yinte Fan and Chao Wang contributed equally to this research.
†Zikan Dong is the corresponding author.

The full name of the authors' affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/26/07
https://doi.org/10.1145/3803437.3805227

Keywords

Rust adoption; Android Open Source Project; Software migration

ACM Reference Format:
Yinte Fan, Chao Wang, Zikan Dong, Tianming Liu, and Haoyu Wang. 2026. An Empirical Analysis of Rust Integration in Android Open Source Project. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3803437.3805227

1 Introduction

Memory-safety vulnerabilities remain a persistent challenge in contemporary mobile systems built on the Android Open Source Project [19], despite ongoing efforts to strengthen the platform's security framework. A recent Android vulnerability report [6] indicates that critical subsystems, such as wireless communication stacks and media processing components, continue to be susceptible to memory corruption attacks. These recurring flaws stem from fundamental characteristics of C and C++, the dominant native languages in Android [45]. Their flexibility enables direct memory manipulation through pointer arithmetic, yet they do not provide automatic bounds checking [52, 62]. As a result, even minor miscalculations in buffer size or offset can trigger severe overflows. Moreover, manual memory management in C/C++ is complex and error-prone; oversights in deallocation or synchronization, for instance, can easily lead to use-after-free or double-free vulnerabilities.

To effectively address these inherent safety shortcomings without incurring substantial performance costs, adopting memory-safe programming languages has emerged as a promising strategy [3, 13, 39, 40, 44]. Among these, Rust stands out due to its unique combination of compile-time memory safety guarantees and seamless interoperability with existing C codebases [34, 36, 38, 42, 58]. This native-level compatibility enables incremental adoption within systems like AOSP while substantially reducing the classes of vulnerabilities that are common in C/C++ codebases.

In April 2021, Google officially announced the introduction of the Rust programming language into AOSP, marking a pivotal moment in Android's evolution [1, 2]. Since then, Rust integration has followed a deliberate and progressive path [4, 14]. Initial deployments

targeted critical subsystems, such as virtualization frameworks and the Binder IPC mechanism, areas where high security and stability are paramount and aligned well with Rust's strengths. Adoption has grown steadily, and today nearly 70 AOSP components incorporate Rust, with usage continuing to expand as developers adopt it for new development and incremental refactoring of legacy modules.

However, the actual scale, security outcomes, integration patterns, and unsafe governance of this large-scale migration remain poorly understood. A systematic study is therefore needed to clarify these aspects, which is essential not only for assessing Rust's effectiveness but also for deriving transferable best practices. Moreover, Rust's coexistence with legacy C/C++ introduces new risks at interoperability boundaries that also warrant systematic study. To fill this knowledge gap, we conduct **the first large-scale, longitudinal empirical study of Rust in AOSP**. Our study covers over 10,000 Rust source files (as of October 2024), 16 versions of the AOSP codebase, 5,375 vulnerability records from the Android Security Bulletin, and 20,999 SAFETY comments associated with unsafe code. Our investigation is guided by four research questions:

RQ1: What is the status of Rust adoption in AOSP? We analyze the distribution of Rust code and then use quarterly snapshots of the AOSP source tree to study its evolution. We observe that: (1) AOSP's Rust footprint is dominated by ecosystem and tooling code, suggesting that establishing the Rust supply chain and toolchain is a prerequisite for in-tree Rust migration. (2) Rust adoption in AOSP shows stepwise growth aligned with major Android releases.

RQ2: How does Rust adoption relate to memory-safety vulnerabilities in AOSP? To address this question, we employ a three-step approach: aligning vulnerabilities with Rustified components to reveal adoption drivers, comparing pre- and post-adoption vulnerability counts to assess security impact, and analyzing residual hotspots in non-Rust code to identify migration priorities. Our analysis yields three main findings corresponding to these three aspects: (1) Rust adoption in AOSP is risk-driven, prioritizing components with a history of frequent memory-safety vulnerabilities. (2) After adoption, most Rustified components exhibit no publicly observed memory-safety vulnerabilities. (3) Memory-safety risk is highly concentrated in a small number of non-Rust regions, such as /frameworks/av and /system/bt, making them prime candidates for future Rust migration.

RQ3: How does AOSP integrate Rust in a mixed-language platform? AOSP is a long-lived, mixed-language system, so Rust's safety benefits depend on how well it integrates with existing build infrastructure and language boundaries. To investigate this, we first analyze the AOSP build system source code to understand how Rust is incorporated, and then examine the mechanisms used for cross-language interaction and the rationale for their selection. Our key observations are: (1) For platform-scale integration, it is effective to treat the platform build system as the source of truth, using Cargo primarily to extract upstream crate metadata. (2) AOSP Rustification follows a *platform-first, automation-assisted* approach: Rust is adopted by reusing existing interoperability boundaries, with automated tools generating bindings.

RQ4: How does AOSP govern unsafe code through SAFETY comments? In AOSP, system developers annotate unsafe code with SAFETY comments to document their safety justifications. We study this governance mechanism by analyzing the quality, clarity, and patterns of these comments across the codebase. We find that AOSP's governance exhibits two predominant patterns. First, safety contracts are rarely enforced through internal checks, resulting in greater reliance on external verification. Second, the effectiveness of a SAFETY comment hinges on its specificity: actionable comments precisely link safety predicates to named entities, whereas anti-patterns, characterized by vague or insufficient operational details, undermine the safety contract and create critical governance gaps.

In summary, our work presents these contributions:

- **Longitudinal, component-level map of Rust adoption in AOSP.** Across 16 quarterly snapshots from Jan. 2021 to Oct. 2024, we construct the first component-by-time view of the in-tree Rust footprint, covering 1,387 AOSP-maintained Rust files across 67 components.
- **Adoption-aligned evidence on memory-safety outcomes.** Using 5,375 Android Security Bulletin entries, including 1,953 publicly disclosed cases, we align vulnerabilities with components and analyze memory-safety trends before and after Rust first appears in each component, while also identifying remaining non-Rust hotspot regions.
- **Empirical account of platform-first Rust integration in AOSP.** We characterize how Rust is built under Soong and how components interact across language boundaries through mechanisms such as IDL/IPC, bindgen, C++ bridges, the C ABI, and JNI.
- **Large-scale study of SAFETY comments in AOSP.** We present the first analysis of 20,999 SAFETY comments paired with unsafe occurrences to estimate contract prevalence and distill best-practice templates and anti-patterns.

2 Background

2.1 Rust Fundamentals

Rust is a systems programming language designed to provide strong safety guarantees, with a particular focus on memory safety. These guarantees are primarily enforced at compile time through Rust's ownership-based memory model. To provide the necessary background for our study, we briefly introduce three core Rust concepts: ownership, lifetimes, and the unsafe keyword.

2.1.1 Ownership. In Rust, ownership specifies which program entity has exclusive control over a value and governs the value's lifetime, including when it may be accessed and when its memory is reclaimed. In contrast, languages such as C and C++ rely on explicit, manual memory management, placing the responsibility for correct allocation and deallocation on the programmer and making such code more prone to errors, including memory leaks, dangling pointers, and use-after-free vulnerabilities. Rust instead enforces ownership rules at compile time, guaranteeing that each value has a single owner and that resources are automatically released when ownership ends, thereby eliminating entire classes of memory-management errors [18, 37, 64].

In addition to ownership itself, Rust regulates how ownership is transferred or temporarily shared through two fundamental operations: moving and borrowing [12, 41, 45, 63].

- **Moving** transfers ownership of a value from one variable to another. After a move, the original variable becomes invalid and can no longer be used, ensuring that ownership remains unique.
- **Borrowing** allows a variable to access a value without taking ownership, enabling temporary sharing under strict constraints. Rust distinguishes two forms of borrowing: (1) Immutable borrowing grants read-only access. Multiple immutable references may coexist, allowing safe concurrent reads. (2) Mutable borrowing grants exclusive, writable access. While a mutable reference is active, no other references are permitted. This restriction prevents conflicting accesses and data races [26].

2.1.2 Lifetime. Lifetimes are a compile-time mechanism used by the Rust compiler to track how long references remain valid. This analysis is performed by the borrow checker, a compiler component that enforces Rust's ownership and borrowing rules to prevent invalid memory accesses. A reference is only permitted to exist within the lifetime of the value it refers to. Accordingly, in Rust, a value's lifetime is tied to ownership: it begins when the value is created and ends when ownership is relinquished or the value is deallocated. By enforcing lifetime constraints at compile time, Rust ensures that references are never used beyond the validity of the underlying data, thereby preventing dangling references and related memory-safety errors [17, 53, 54, 60].

2.1.3 Unsafe Keyword. Rust's core memory-safety guarantees are enforced at compile time through its ownership and lifetime system. However, not all low-level operations can be expressed or verified within these guarantees. In particular, operations that interact directly with raw memory or cross language boundaries may lack the lifetime and aliasing information required by the compiler for full verification. For example, dereferencing a raw pointer is disallowed in safe Rust because raw pointers do not carry sufficient guarantees about validity, alignment, or borrowing discipline.

To support such low-level use cases while preserving overall safety, Rust provides the `unsafe` mechanism as a controlled escape hatch. Rather than disabling safety checks globally, `unsafe` allows developers to explicitly opt into a small set of operations whose correctness must be justified by additional invariants beyond what the compiler can enforce [35, 43, 49, 50]. Specifically, `unsafe` permits exactly five classes of operations that bypass certain compile-time checks [16]: (1) dereferencing raw pointers, (2) calling unsafe functions, (3) accessing or modifying mutable static variables, (4) implementing unsafe traits, and (5) accessing fields of unions [29, 30].

2.2 Infrastructure for Rust Integration in AOSP

To support Rust development within AOSP, Android provides a set of platform-specific tools and integration mechanisms. This section briefly introduces the build-system support and cross-language interoperability mechanisms used in practice. A detailed analysis of their design choices and implications is provided in Section 5.

2.2.1 Build System Integration. AOSP adopts Soong [7, 15] as its platform build system, whereas the Rust ecosystem typically relies on Cargo [11]. Beyond differences in manifest formats (Blueprint `Android.bp` [5] versus `Cargo.toml` [20]), the two serve distinct roles. In AOSP, Soong modules define the platform-enforced and reviewable build contract, while Cargo describes crate structure and dependency relationships in the Rust ecosystem.

- **Soong.** Soong is Android's platform build system and serves as the authoritative source of truth for module definitions, build variants, and dependencies. It supports multi-language builds and relies on explicitly vendored and pinned dependencies.
- **Cargo.** Cargo is Rust's standard build tool and package manager. It defines crates and dependency graphs via `Cargo.toml` and supports common Rust workflows. In AOSP, Cargo is mainly used to interpret upstream crate structure and metadata, while Soong remains the canonical build specification.

2.2.2 Cross-language Interoperability. As a mixed-language platform, AOSP requires Rust components to interoperate with existing Java, C, and C++ code through established interfaces [48]. To this end, AOSP supports several interoperability mechanisms that enable Rust integration without disrupting existing components.

- **IDL/IPC.** Android relies on IPC and interface definition mechanisms to decouple components across processes and layers. Interfaces such as Android Interface Definition Language (AIDL) and proto-style schemas define stable contracts that remain agnostic to the implementation language. AOSP reuses these existing boundaries, allowing Rust implementations to replace or coexist with native services behind unchanged IPC contracts, without requiring modifications on the client side [23].
- **Bindgen.** Bindgen automatically generates Rust bindings from C headers, enabling Rust code to reuse C/C++ APIs. This approach reduces manual binding effort, but shifts responsibility to managing unsafe interactions at the generated boundary [24].
- **Cxx::bridge.** Cxx::bridge provides a structured Rust–C++ interoperability mechanism based on explicitly defined shared interfaces. Compared to raw bindings, it centralizes glue code and helps make unsafe boundaries more explicit and reviewable [25].
- **C ABI / FFI Primitives.** For low-level integration, Rust exposes and consumes C-compatible interfaces via the C ABI and FFI primitives such as `extern "C"`, `CString`, and raw pointers. In these cases, safety relies on clearly specified cross-language contracts that define the obligations of non-Rust callers [27].
- **JNI.** Rust can also interoperate with Java through JNI, although this mechanism is less commonly used in AOSP compared to native or service-layer integration [28].

3 RQ1: What Is the Status of Rust Adoption in AOSP?

This section establishes the empirical baseline of Rust adoption in AOSP. To this end, we quantify the in-tree Rust footprint, examine its distribution across the repository structure, and track its growth over time using quarterly snapshots.

3.1 Distribution Across AOSP Components

To characterize where Rust is being adopted in AOSP, we examine its distribution at two granularities: across top-level directories

(such as `/system` and `/packages`) and within individual components (specific modules such as Bluetooth and Binder). This two-level analysis reveals both the breadth of Rust adoption across AOSP and the depth of integration within particular areas.

3.1.1 Methodology. To analyze the distribution patterns of Rust code within the AOSP source tree, we employ an operational definition based on its directory structure. First, we examine the distribution across top-level directories, which reflect code provenance and management categories (e.g., /external for third-party code, /packages for independently distributable components). Subsequently, to pinpoint Rust integration into specific functional modules more granularly, we group all in-tree .rs files into 67 components based on their directory paths, with each component manually verified.

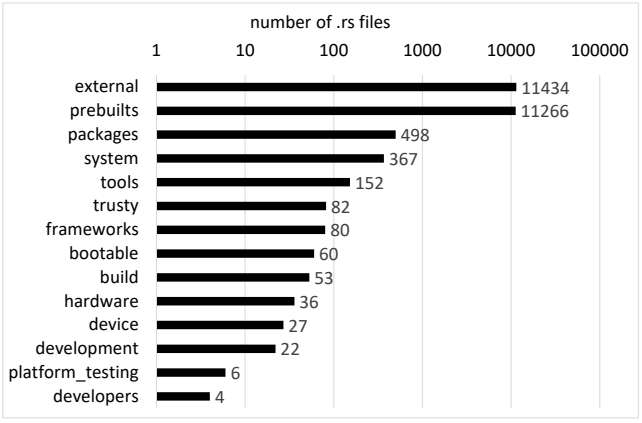


Figure 1: Count of Rust files in different top-level directories.

3.1.2 Results. As shown in Figure 1, the full AOSP source tree contains 24,087 `.rs` files, but most are under `/external` and `/prebuilts`, which host third-party code and toolchain artifacts. We therefore exclude these directories when characterizing AOSP-maintained adoption, leaving 1,387 in-tree `.rs` files for the analyses below.

Among AOSP-maintained Rust files, the distribution across top-level directories is uneven. Rust code concentrates in a few hotspots, such as `/packages` and `/system`, while many other directories contain only tens of Rust files. Notably, some directories with extensive native codebases, like `/frameworks`, have relatively few Rust files. This pattern suggests that Rust adoption is driven by localized introductions rather than large-scale rewrites. The selective nature of Rust integration will be further explored in the following sections.

Observation #1: AOSP's Rust footprint is dominated by ecosystem/tooling code, suggesting that building the Rust supply chain and toolchain is an important foundation for in-tree Rust migrations. Also, Rust is selectively introduced, rather than through large-scale rewrites of existing native code.

Component-level Adoption. To further zoom into component-level adoption, we group Rust files into 67 components by directory location and measure Rust-only code volume. Eight components exceed 10KLoC, as shown in Figure 2, and the two largest are Bluetooth and Virtualization under `/packages/modules`. This directory

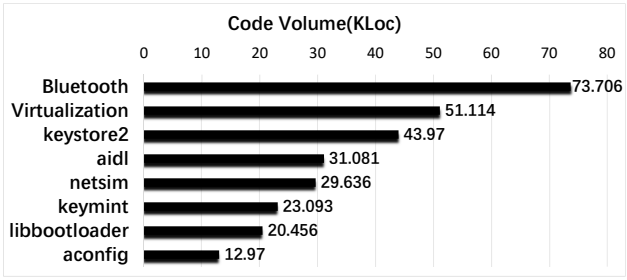


Figure 2: Top 8 largest components of Rust code.

Table 1: Classification of AOSP Rust components.

Category	Example	Count
Security & Trusted Computing		**24**
Security Foundation	hwwsk; keymint; …	16
System Safeguards	mls; fsverity; …	8
Communication & Frameworks		**20**
Communication Protocol Stack	binder; nfc; …	10
Core Service Frameworks	input; apexsupport; …	10
Development & Support		**15**
Development Toolchain	aconfig; cargo-embargo; …	11
System Virtualization	cuttlefish; virtualization; …	4
Hardware Resource Control		**8**
Hardware Abstraction Layer	automotive; light	2
Graphics & Multimedia	nativewindow; gui; …	6
Grand Total		**67**

hosts modular system components that can be updated more independently from the core OS, making it a practical landing zone for introducing Rust into large codebases.

Functional Classification of Rustified Components. To understand what roles these components play, we classify all 67 components into four functional domains (Table 1):

- **Security & Trusted Computing** (24 components): cryptographic services, key management, and system integrity enforcement (e.g., `keymint`, `keystore2`).
- **Communication & Frameworks** (20 components): IPC mechanisms, protocol stacks, and core service frameworks (e.g., Binder, Bluetooth, NFC).
- **Development & Operational Support** (15 components): build tools, testing infrastructure, and virtualization support (e.g., `car go-embargo`, Cuttlefish).
- **Hardware Resource Control** (8 components): HAL implementations and multimedia subsystems (e.g., `nativewindow`, `gui`).

Observation #2: Two-thirds of Rustified components (44/67) handle security-critical or IPC/framework logic, indicating that Rust adoption targets areas where memory safety violations pose the highest risk.

Strategic Drivers Behind Component Selection. Our analysis reveals two key factors influencing which components adopt Rust:

(1) Security Exposure and Attack Surface. Components in "Security & Trusted Computing" and "Communication & Frameworks" handle sensitive data, external inputs, or cross-process communication, all of which are high-risk areas for memory corruption vulnerabilities. Their prevalence (44/67) suggests that reducing attack surface drives adoption decisions.

(2) Deployment Feasibility and Modularity. The largest Rust components (Bluetooth, Virtualization) reside in `/packages/modules`, which supports independent updates without full system rebuilds. This architectural decoupling lowers the engineering friction of migrating large, complex codebases to Rust.

Engineering Patterns in Rust Integration. Beyond functional domains, we observe four distinct engineering patterns:

- **Core Rewrites**: replacing memory-unsafe implementations in critical paths (e.g., `keystore2`).
- **Boundary Hardening**: HAL implementations and input-processing layers interfacing with hardware or untrusted sources.
- **Safe Wrappers**: encapsulating existing C/C++ libraries to isolate unsafe code within well-defined seams (e.g., Binder bindings).
- **Testing Infrastructure**: building fuzzers and test harnesses in Rust to improve safety tooling around legacy components.

Observation #3: AOSP employs Rust strategically, not as a wholesale replacement, but as a surgical tool applied where memory safety, external exposure, or testing demands justify the migration cost.

3.2 Adoption Timeline

To understand when and how Rust was introduced into AOSP, we track its growth trajectory from early adoption to the present state.

3.2.1 Methodology. We take quarterly snapshots of the AOSP source tree from January 2021 to October 2024, yielding 16 data points across this period. For each snapshot, we count (1) the number of Rustified components as defined in Section 3.1, and (2) the total number of `.rs` files repository-wide, including those in `/external` and `/prebuilts`.

3.2.2 Results. Figure 3 reveals a consistent upward trend: Rustified components grow from near zero in early 2021 to over 60 by October 2024, while the total `.rs` file count increases proportionally. However, growth is not uniform. We observe step-like jumps coinciding with major Android release cycles (e.g., Android 14 to Android 15), with more gradual increments during interim development periods. This pattern suggests that substantial Rust integration is concentrated in release windows rather than distributed continuously throughout the development cycle.

Observation #4: Rust adoption in AOSP exhibits stepwise growth aligned with major Android releases, indicating that large-scale Rust integration is batched into release cycles rather than merged incrementally.

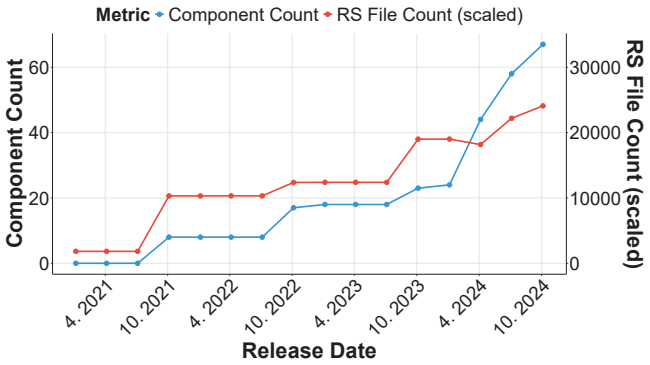


Figure 3: Rust adoption timeline in AOSP from Jan. 2021 to Oct. 2024 based on quarterly snapshots.

4 RQ2: How Does Rust Adoption Relate to Memory-safety Vulnerabilities in AOSP?

In this section, we examine how the introduction of Rust relates to the distribution and evolution of memory-safety vulnerabilities in AOSP. We begin by mapping Android vulnerability records to individual system components. Building on this alignment, we analyze how memory-safety vulnerability counts change before and after Rust adoption. Finally, we identify remaining concentration points of memory-safety vulnerabilities in non-Rust code, highlighting components that may benefit from future Rust migration.

4.1 Pre-adoption Risk Profile of Rustified Components

We first examine whether the selection of Rustified components is consistent with their historical memory-safety risk. To this end, we count, for each Rustified component, the number of vulnerabilities reported prior to its Rustification.

4.1.1 Methodology. To identify memory-safety vulnerabilities in AOSP, we first collect 5,375 vulnerability records from the Android Security Bulletin (Aug. 2015 to Oct. 2024) [6], including 1,953 publicly disclosed cases. We then apply an LLM-assisted labeling pipeline to classify memory-safety issues, configuring the model with a temperature of 0. To assess label quality, we manually validate a random sample of 100 records, of which 89 are correctly labeled, providing a sanity check for the automated results. Finally, we align the identified cases with Rustified components by matching vulnerability file path prefixes to the corresponding component directories.

4.1.2 Results. Among the 67 Rustified components identified in Section 3, only 14 have publicly observable vulnerability records in our dataset. This limited coverage is expected for two reasons. First, public vulnerability disclosures provide an incomplete view of issues addressed in AOSP, as many vulnerabilities are fixed internally or never publicly reported. Second, Rust is sometimes introduced proactively into security-sensitive components, such as trusted services and protocol parsers, even when these components lack an extensive public vulnerability history.

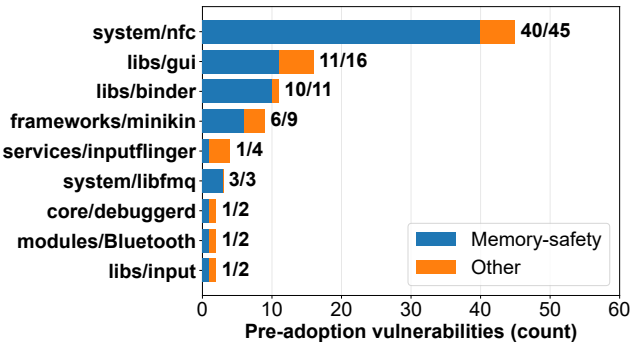


Figure 4: Pre-adoption vulnerability burden of Rustified components with memory-safety history.

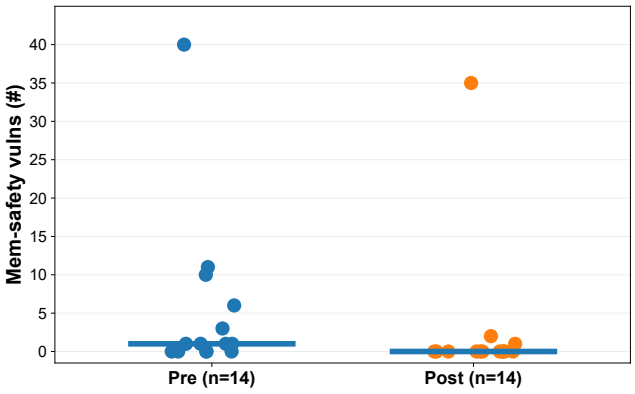


Figure 5: Memory-safety vulnerability counts per Rustified component before and after T_0. Each dot is one component; horizontal bars denote medians.

We therefore restrict our analysis to these 14 components. Across these components, we identify 96 vulnerabilities reported prior to Rustification, of which 74 are memory-safety issues (77.1%). By comparison, memory-safety vulnerabilities account for only 37.4% of the entire dataset. As shown in Figure 4, these pre-Rustification risks are highly concentrated: NFC, Binder, and GUI together account for 61 of the 74 memory-safety vulnerabilities (82.4%).

Observation #5: Rust adoption in AOSP appears risk-driven: components with a history of frequent memory-safety vulnerabilities are prioritized, and security-critical boundary components are Rustified as a form of preventive hardening.

4.2 Overall Shift in Memory-safety Vulnerability Counts (Pre vs. Post Adoption)

To assess how memory-safety vulnerability counts change with Rust adoption, we compare the number of reported memory-safety vulnerabilities before and after the component becomes Rustified, and denote this Rustification time as (T_0).

4.2.1 Methodology. Using the timeline data from Section 3.2, we determine the Rustification time T_0 for each component. We then combine this information with the vulnerability dataset from Section 4.1 to count the number of memory-safety vulnerabilities reported before and after T_0.

4.2.2 Results. Figure 5 shows a pronounced shift toward zero memory-safety vulnerabilities after Rust adoption. Specifically, 11 of the 14 components (78.6%) reported no memory-safety vulnerabilities after T_0, and the median count decreased from 1 before adoption to 0 afterward. At the same time, a small number of components continue to exhibit non-zero vulnerability counts after T_0 (e.g., Bluetooth). It is worth noting that components are Rustified at different points in time, resulting in varying lengths of post-adoption observation periods across components. Accordingly, we interpret absolute pre- and post-adoption vulnerability counts with caution and view the observed shift toward zero as suggestive rather than conclusive evidence.

Post-adoption Outliers. While most components exhibit a clear reduction in memory-safety vulnerabilities following Rust adoption, a small number of exceptions persist. Notably, in Bluetooth, the number of documented memory-safety vulnerabilities increases from 1 before T_0 to 35 after T_0. In the case of Binder, although the total vulnerability count decreases from 10 to 1, a memory-safety issue is still observed after T_0. Two factors help explain these outliers. First, several large components are only partially migrated: Rust replaces selected modules, while substantial legacy C/C++ code remains on exposed paths (e.g., Bluetooth and NFC). Second, the Rustification time T_0 marks the initial introduction of Rust code and may precede the point at which Rust becomes the default implementation on the critical path. For example, in Binder, post-adoption vulnerabilities cluster close to T_0, which is consistent with issues originating in pre-existing code or identified before the Rust implementation is fully rolled out. Overall, these outliers suggest that the security benefits of Rust depend on how completely it replaces attack-facing code within a component.

Observation #6: After Rust adoption, the majority of Rustified components exhibit *no publicly observed memory-safety vulnerabilities*; the remaining cases are concentrated in components with partial Rust coverage or very recent migrations.

4.3 Remaining Memory-safety Hotspots and Migration Opportunities

Our preceding analysis indicates that Rust adoption in AOSP is largely risk-driven. Based on this observation, we next examine how memory-safety vulnerabilities are distributed in non-Rust code and identify components that may benefit from further Rustification.

4.3.1 Methodology. To localize remaining memory-safety risk in non-Rust code, we group vulnerability records by coarse-grained code regions defined using the first three levels of the source-tree directory hierarchy (e.g., `/frameworks/av/media`). We then rank these code regions by the number of associated memory-safety vulnerabilities, which allows us to identify stable and actionable hotspot areas rather than fragmented, file-level signals.

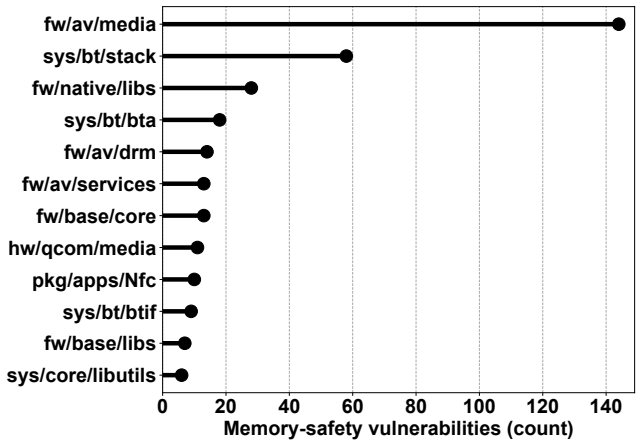


Figure 6: Top non-Rust hotspot code regions.

4.3.2 Results. Figure 6 shows that the remaining observable memory-safety vulnerabilities are highly concentrated in a relatively small number of code regions. In particular, several of the highest-ranked regions fall under the broader `/frameworks/av` hierarchy, suggesting that memory-safety risk in the media stack spans multiple subdirectories rather than being isolated to a single module. Additional hotspots appear in boundary-facing components, such as Bluetooth-related paths (e.g., `/system/bt/*`), as well as in core native libraries (e.g., `/frameworks/native/*`). This concentration naturally suggests a phased migration strategy: prioritizing the highest-ranked regions first, and then drilling down further within each region to the specific components responsible for most reported memory-safety issues.

Observation #7: Memory-safety risk is concentrated in a small number of non-Rust regions, making these regions natural candidates for prioritization in future Rust migrations.

5 RQ3: How Does AOSP Integrate Rust in a Mixed-language Platform?

In this section, we examine how Rust is integrated into AOSP and how it coexists with legacy C/C++ and other languages within a mixed-language platform. Specifically, we study the platform-scale toolchain and build integration for Rust, analyze the mechanisms used for cross-language interoperability, and examine how unsafe code is governed through SAFETY comments.

5.1 Build System Integration: Rust Compilation and Linking in Soong

To understand how Rust is integrated into the AOSP build system, we analyze the Soong source code and the structure of `Android.bp` files. Our analysis indicates that Rust follows a Soong-governed build workflow, in which `Android.bp` serves as the canonical build specification, rather than Cargo acting as the primary build driver.

In this design, Rust code is represented as explicit Soong modules. This is reflected in Soong's Rust front-end, which defines a family of `rust_*` module types corresponding to standard Rust targets, including libraries (`rust_library`), executables (`rust_binary`), procedural macros (`rust_proc_macro`), and test targets (`rust_test`). By treating Rust as a first-class language within Soong, Rust components participate in the same module graph and build process as the rest of the platform.

To incorporate third-party Rust crates while preserving Soong as the source of truth, AOSP provides the `cargo_embargo` tool, which translates Cargo projects into equivalent Soong module definitions [8]. Specifically, `cargo_embargo` executes a Cargo build for a given crate, extracts the underlying compilation commands (e.g., `rustc` and `cc` invocations), and uses this information to generate corresponding `Android.bp` rules. The resulting modules, together with pinned metadata, are then vendored under `/external/rust`.

Observation #8: For large, platform-scale systems such as AOSP, Rust integration benefits from treating the platform build system as the source of truth, while using Cargo primarily to extract upstream crate metadata and build structure.

Release-driven Dependency Updates. We observe batch update scripts such as `regenerate_all.sh` in `cargo_embargo`, indicating that third-party Rust dependencies are refreshed through coordinated updates rather than ad hoc changes.

5.2 Cross-language Interoperability Mechanisms and Patterns

Next, we examine how Rust interacts with other languages in AOSP, with a focus on C/C++ and Java. To this end, we identify the primary cross-language integration mechanisms used by Rustified components and analyze the engineering trade-offs they entail, including integration effort, interface stability, and the extent to which residual unsafety remains at language boundaries.

5.2.1 Methodology. To analyze cross-language interoperability, we examine all Rustified components to identify their primary interaction mechanisms. We determine each component's mechanism using a combination of automated pattern matching (e.g., regex-based detection) and manual verification to ensure accuracy.

Based on this analysis, we classify each component into one of the following seven categories: (1) **IDL/IPC**: Contract-based interfaces, including AIDL and proto-based interfaces. (2) **Bindgen**: Automatically generated Rust bindings derived from C/C++ headers. (3) **Cxx::bridge**: Structured Rust–C++ interoperability using explicitly defined shared interfaces and paired glue code, with a well-defined unsafe boundary. (4) **JNI**: Rust–Java interaction via the Java Native Interface. (5) **C ABI**: Minimal-runtime Rust components (e.g., `no_std`) exposed through C-compatible ABI wrappers. (6) **Hybrid**: Components that combine multiple interoperability mechanisms, such as IDL/IPC together with bindgen. (7) **None**: Components that do not expose an explicit cross-language interface.

5.2.2 Results. Figure 7 summarizes the distribution of cross-language interoperability mechanisms across Rustified components. The results show that cross-language interaction is the norm, with only a small fraction of components falling into the *None* category. Among

the explicit mechanisms, IDL/IPC and bindgen are the most prevalent. By contrast, JNI is used by only two components, indicating that Rust rarely interacts directly with Java code in AOSP.

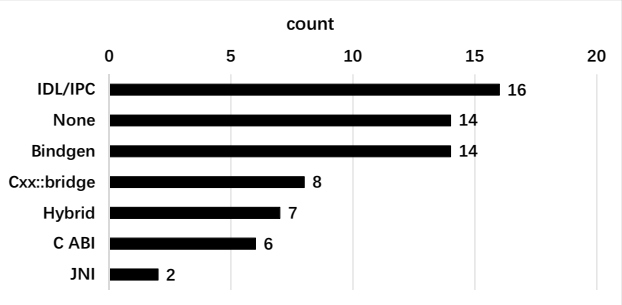


Figure 7: Interoperability mechanisms of 67 Rustified components in AOSP.

Cross-language Interaction Patterns and Rationale of Selection. Building on the overall distribution, we examine how interoperability mechanisms are implemented in AOSP and the rationale behind their selection.

`AIDL/IPC-based` integration is the most prevalent pattern, building on AOSP's longstanding cross-process communication foundation, the AIDL. To integrate Rust, developers simply set `rust.enabled` in an aidl_interface module, and the Soong build system automatically generates the necessary bindings. Its dominance stems from this ability to leverage existing, stable service boundaries with minimal disruption.

`Bindgen` is a command-line tool for automatically generating Rust FFI bindings from C/C++ headers. In AOSP, developers declare a `rust_bindgen` module in `Android.bp`, which triggers Soong to invoke bindgen and produce the corresponding bindings. This mechanism is widely adopted because it automates interaction at scale. The main trade-off, however, is the need to carefully document and encapsulate the resulting localized unsafety. Overall, the prevalence of both IDL/IPC and bindgen highlights the value of automation and their deep integration with the Soong build system.

The `Cxx::bridge` mechanism requires developers to explicitly define shared interfaces in a dedicated Rust module, which then generates paired glue code for both Rust and C++. This mechanism is chosen when stronger type guarantees are needed for exchanging complex data structures between languages.

The `C ABI` provides a lightweight interoperability mechanism by exposing Rust functions as C-compatible interfaces using `#[no_std]` and `extern "C"`. It is chosen for low-level or performance-critical scenarios, like embedded components, where minimal runtime overhead is essential.

`JNI` enables Rust–Java interaction through standard JNI conventions, with type conversions explicitly handled via the `JNIEnv` interface. However, this mechanism is rarely used in AOSP, reflecting Rust's current primary role in system-level services rather than in application-layer components, which are typically Java-based.

Based on the analysis of Cross-language interaction patterns and rationale of selection above, we summarize cross-language integration strategies as follows.

Platform-first Cross-language Integration Strategies. Rather than introducing new interoperability paths, Rust integration in AOSP generally follows two guiding principles: aligning with existing platform-level cross-language mechanisms to minimize ecosystem disruption, and relying on automated code generation to support integration at scale. Based on these principles, we summarize three reusable integration strategies for developers integrating Rust into mixed-language platforms:

- **Contract-first Platform Integration.** For components behind well-defined service boundaries, the platform's established contract mechanisms should be preferred. Interface definitions such as IDL or RPC schemas provide stable integration anchors for introducing Rust without affecting clients.
- **Wrapping-based Legacy Reuse.** When interfacing with existing C/C++ codebases, automated binding generators provide a scalable integration approach. *bindgen* is suitable for broad coverage of large or evolving APIs, while *cxx::bridge* supports structured integration when stronger type guarantees are required.
- **Minimal ABI Integration.** For low-level or constrained environments, Rust components can expose a minimal `C ABI` surface. This approach prioritizes runtime efficiency and relies on explicit safety contracts to define the obligations of C callers.

Observation #9: AOSP Rustification follows a *platform-first, automation-assisted* approach: Rust is adopted by reusing existing interoperability boundaries, with automated tools to create bindings. Under this model, the security benefits depend on how effectively these boundaries confine residual unsafety.

6 RQ4: How Does AOSP Govern Unsafe Code Through SAFETY Comments?

In Rust, an `unsafe` block is a language construct that allows operations bypassing the compiler's memory safety guarantees, such as dereferencing raw pointers or calling foreign functions. While necessary for low-level system programming, these blocks require programmers to manually uphold Rust's safety invariants.

While RQ3 (Section 5.2) reveals that extensive cross-language interaction introduces many unsafe boundaries, RQ2 (Section 4.2) nonetheless demonstrates that Rust adoption has been highly successful in reducing memory-safety vulnerabilities. This success motivates us to examine how AOSP manages the inherent risks of unsafe code. We observe that AOSP developers have widely adopted the practice of annotating unsafe blocks with SAFETY comments, following Rust's official best practices [10, 31, 32]. Given this widespread adoption and AOSP's demonstrated governance maturity, we first assess the quality and clarity of SAFETY comments in practice (Section 6.1), and then distill recurring best practices and anti-patterns from SAFETY comment writing (Section 6.2).

6.1 Assessing SAFETY Comment Quality and Clarity

Given the central role of SAFETY comments in governance, this section presents a systematic evaluation of their application. We analyze a large corpus of comments along three key dimensions:

their quality, the attribution of safety responsibilities, and the types of unsafe operations they justify.

6.1.1 Methodology. We extract 20,999 SAFETY comments from AOSP's Rust codebase using pattern matching on `//` and `#` comments containing "SAFETY", capturing each comment with ten subsequent lines of code for context. From this corpus, we draw a stratified random sample of 1,000 comments. Each comment is annotated along three dimensions: (1) **Quality:** `CLEAR` (precise, verifiable predicates), `VAGUE` (ambiguous or insufficient detail), or `TODO` (explicitly incomplete). (2) **Responsibility:** `ENV_ASSUMPTION` (external guarantees), CALLER_OBLIGATION (caller preconditions), `INTERNAL_CHECK` (local validation), or `UNCLEAR` (insufficient evidence). (3) **Operation Type:** Categorized by Rust's five unsafe capabilities (Section 2.1.3).

Annotations are performed using an LLM prompted with schema definitions and examples. Manual validation of 100 random samples yields 89% agreement on quality, 88% on responsibility, and 96% on operation type.

6.1.2 Results. Table 2 summarizes the distribution with 95% Wilson confidence intervals [22].

Comment Quality. While 76.5% of comments are `CLEAR`, 23.5% remain `VAGUE`, lacking actionable predicates.

Responsibility Attribution. The largest category is `UNCLEAR` (36.5%), reflecting observability limits in our ten-line context window rather than necessarily poor documentation. Among identifiable cases, external assumptions (26.0%) and caller obligations (23.8%) dominate, while internal checks constitute only 13.7%.

Operation Types. Unsafe calls dominate at 79.5%, followed by raw pointer dereferences (15.6%). Unsafe trait implementations, union accesses, and mutable statics collectively account for <5%.

Observation #10: AOSP's unsafe code exhibits two patterns: (1) *Call-based dominance*: 79.5% of unsafe operations are function calls versus 15.6% pointer manipulation. (2) *Externalized verification*: only 13.7% of contracts reference internal validation, while 49.8% rely on caller obligations or environmental assumptions, shifting the verification burden away from the code itself.

Axis	Category	Count	% (95% Wilson CI)
`Quality`	CLEAR	765	76.5% [73.8%, 79.0%]
	VAGUE	235	23.5% [21.0%, 26.2%]
	TODO	0	0.0% [0.0%, 0.4%]
`Responsibility`	UNCLEAR	365	36.5% [33.6%, 39.5%]
	ENV_ASSUMPTION	260	26.0% [23.4%, 28.8%]
	CALLER_OBLIGATION	238	23.8% [21.3%, 26.5%]
	INTERNAL_CHECK	137	13.7% [11.7%, 16.0%]
`Operation Type`	UNSAFE_CALL	795	79.5% [76.9%, 81.9%]
	RAW_POINTER	156	15.6% [13.5%, 18.0%]
	UNSAFE_TRAIT	48	4.8% [3.6%, 6.3%]
	UNION_ACCESS	1	0.1% [0.0%, 0.6%]
	MUT_STATIC	0	0.0% [0.0%, 0.4%]

Table 2: Prevalence in sample ($N = 1000$). Percentages with 95% Wilson confidence intervals.

6.2 Extracting Best Practices and Anti-Patterns

Building on the preceding assessment of SAFETY comment quality and clarity, we further analyze the annotated corpus to identify recurring patterns associated with effective and ineffective SAFETY comments. Based on this analysis, we derive a taxonomy that systematically categorizes these patterns observed in the codebase.

6.2.1 Methodology. To systematically characterize effective and ineffective safety documentation, we develop a taxonomy comprising five best-practice patterns and three anti-patterns. Best-practice patterns are organized by the type of safety guarantee they articulate: (1) **Caller Checklist**: Enumerates preconditions the caller must satisfy. (2) **Internal Enforcement**: References local validations (e.g., assertions, guards) performed within the function. (3) **Environment Assumption**: Delegates responsibility to trusted external entities (e.g., OS guarantees, FFI contracts). (4) **Layout/Type Justification**: Argues layout or type compatibility for unsafe casts or transmutations. (5) **Bounds/Length Constraints**: States numeric predicates justifying pointer arithmetic or slice construction.

Anti-patterns capture comments where safety guarantees are unclear or non-actionable: (1) **Safe Without Conditions**: Asserts safety without specifying verifiable predicates (e.g., "This is safe"). (2) **Pronoun Vagueness**: Uses ambiguous referents ("it", "this") without naming the justified entity. (3) **Traceability Gaps**: References to external checks that cannot be localized.

We identify candidate comments using keyword-based heuristics. For best practices, we search for concrete predicates (e.g., "non-null", "bounds") and responsibility markers ("caller must", "we checked"). For anti-patterns, we flag unqualified safety claims and vague pronouns lacking predicates. We rank candidates by specificity scores and select the top 25 per category (200 comments in total). Manual review confirms classifications and refines pattern templates.

6.2.2 Results. We distill five reusable templates for best practices based on our taxonomy, using standardized placeholders: `<entity>` (variables/functions/types), `<predicate>` (conditions such as non-null or aligned), `<unsafe>` (operations), `<external>` (trusted components), and `<TypeA>`/`<TypeB>` (type conversions).

(1) Caller Checklist. Assigns preconditions to the caller. *Template:* `Caller must ensure <entity> satisfies <predicate>.` *Example:* "Caller must ensure `ptr` is non-null and aligned."

(2) Internal Enforcement. References local validations performed earlier. *Template:* `We checked <predicate> above, so <unsafe> is valid.` *Example:* "We verified `len <= capacity` at line 42, so pointer arithmetic is safe."

(3) Environment Assumption. Delegates to external guarantees. *Template:* `<external> guarantees <predicate>, thus <unsafe> is valid.` *Example:* "Kernel guarantees this fd remains open during execution."

(4) Layout/Type Justification. Argues type or layout compatibility. *Template:* `<TypeA> is layout-compatible with <TypeB> (#[repr(C)]).` *Example:* "`MyStruct` uses `#[repr(C)]`, matching C ABI layout."

(5) Bounds/Length Constraints. States numeric invariants for memory safety. *Template:* `<len/range predicates> ensure safe memory access.` *Example:* "`offset < buffer.len()` guarantees in-bounds access."

These best-practice templates provide developers with actionable, reviewable guidelines for justifying unsafe code, moving beyond vague claims to verifiable safety contracts.

6.2.3 Anti-Pattern Catalog. We further identify three anti-patterns that undermine contract clarity.

(1) Safe Without Conditions. Asserts safety without verifiable conditions (e.g., "This is just safe"), providing no actionable justification for reviewers to validate the claim.

(2) Pronoun Vagueness. Employs ambiguous referents without naming the specific justified entities (e.g., "It's valid here"), leaving the safety target unclear.

(3) Traceability Gaps. Shifts responsibility of safety verification to other parts of the system without concrete references (e.g., "Checked elsewhere"), breaking the traceability chain.

These anti-patterns break the contract model by substituting implicit trust for explicit reasoning. While their overall frequency remains low, their presence indicates opportunities for enforcement through linting or review guidelines.

Observation #11: Effective safety comments exhibit high specificity: they bind concrete predicates to named entities with clear responsibility attribution. Anti-patterns, conversely, rely on vague assertions or unverifiable claims, undermining the contract model and creating governance gaps.

7 Discussion

Drawing from the large-scale, real-world experience of Rust adoption in AOSP, we formulate a set of actionable guidelines for integrating Rust into similar complex, mixed-language platforms. These guidelines address the key industrial challenges of risk management, build integration, interoperability, and safety governance.

Invest in Toolchain and Dependency Management as Foundational Work. Early investment in vendoring the Rust toolchain and curating dependencies is a prerequisite for building and reviewing Rust code at platform scale. Treat this as critical infrastructure, not per-component overhead. (Obs. #1)

Prioritize Migrations by Exposure and Historical Vulnerability Density. Focus migration efforts on security-critical, boundary-facing components, and use historical vulnerability data to target the most problematic hotspots first. Evidence-driven prioritization maximizes security return on investment. (Obs. #2, #5, #6, #7)

Adopt a Step-by-step, Component-by-component Migration Approach. Instead of rewriting everything at once, migrate the system one piece at a time, like fixing one room at a time while the rest of the house stays livable. This limits problems to a single component and keeps the overall system stable. (Obs. #3)

Align Migration Cadence With the Platform's Release Lifecycle. Synchronize the introduction of Rust code with the platform's established release and integration windows to streamline testing, validation, and delivery. (Obs. #4)

Measure Success by Critical-path Replacement, Not Initial Adoption. A migration succeeds only when Rust covers security-critical paths. Track residual vulnerabilities in exposed legacy code to gauge rollout effectiveness. (Obs. #6)

Subordinate Cargo to the Platform Build System for Deterministic Control. Use Cargo to extract upstream crate metadata and build graphs, but transform and pin all dependencies as native platform modules. This ensures the build is hermetic, auditable, and conforms to release-engineering requirements. (Obs. #8)

Adopt a Platform-first, Automation-assisted Interoperability Playbook. Prioritize reusing the platform's existing contract boundaries. For legacy code integration, prefer automated tools like `bindgen` for scalable interoperation, while reserving other, more manual mechanisms for a complementary role in specific scenarios where they provide distinct security advantages. (Obs. #9)

Enforce Actionable Safety Contracts for All Unsafe Blocks. Every unsafe block should be justified by a comment that states checkable predicates and makes responsibilities explicit. Avoid non-actionable claims like "safe" or "valid" without operational definitions, as they offer no real assurance. (Obs. #10, #11)

8 Related Work

Rust-related Empirical Study. Prior research has investigated Rust adoption from several perspectives. Early work analyzed memory safety using CVE reports [55, 56, 61]. Later studies examined the use of unsafe across the Rust ecosystem or focused on high-star crates [35, 43, 51]. More recently, attention has turned to Rust's integration within specific domains, such as the Rust-for-Linux initiative from a community perspective [47]. While studies like Rust-for-Linux examine community-driven integration, Rust's adoption in product-oriented industrial platforms like AOSP, with strict release cycles and supply-chain constraints, is less understood. Our work addresses this gap.

Rust in Embedded Systems. Rust's unique blend of C-like performance and strong memory safety guarantees has led to its growing adoption in embedded systems. This is evidenced by industry frameworks such as Tock [33], Embassy [9], and Drone [21], as well as increasing academic attention on its practices and security implications [46, 57, 59]. Our work contributes to this body of knowledge by focusing on AOSP, a large-scale, real-world example of an embedded software platform adopting Rust.

9 Conclusion

We present the first longitudinal, systematic empirical study of Rust in AOSP. Across 16 snapshots from 2021–2024, Rust adoption expands steadily and concentrates in security- and interface-exposed components. Aligning Android Security Bulletin records with component adoption timelines suggests fewer *publicly observed* memory-safety vulnerabilities after Rust is introduced, while remaining risk clusters in a small number of non-Rust hotspot regions. AOSP's Rustification is largely platform-first, reusing existing build and interoperability boundaries; residual unsafety is governed through widespread SAFETY comments, from which we distill actionable best-practice templates and anti-patterns. Overall, this work clarifies how Rust is adopted, integrated, and governed in a safety-critical mobile platform, and provides practical guidance for other large-scale systems pursuing memory-safe migration.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (grant No.62572209) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

References

[1] 2021. Integrating Rust Into the Android Open Source Project. https://security.googleblog.com/2021/05/integrating-rust-into-android-open.html.

[2] 2021. Rust in the Android platform. https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[3] 2021. Rust in the Linux kernel. https://security.googleblog.com/2021/04/rust-in-linux-kernel.html.

[4] 2023. Scaling Rust Adoption Through Training. https://security.googleblog.com/2023/09/scaling-rust-adoption-through-training.html.

[5] 2025. Android Blueprint. https://source.android.com/docs/setup/build/rust/building-rust-modules/android-rust-modules.

[6] 2025. Android Security and Update Bulletins | Android Open Source Project. https://source.android.com/docs/security/bulletin.

[7] 2025. Build overview | Android Open Source Project. https://source.android.com/docs/setup/build.

[8] 2025. Cargo Embargo. https://android.googlesource.com/platform/development/+/main/tools/cargo_embargo.

[9] 2025. Embassy. https://embassy.dev/.

[10] 2025. Fuchsia documentation. https://fuchsia.dev/fuchsia-src/development/languages/rust/unsafe.

[11] 2025. More about Cargo and Crates.io - The Rust Programming Language. https://doc.rust-lang.org/stable/book/ch14-00-more-about-cargo.html.

[12] 2025. References and Borrowing - The Rust Programming Language. https://doc.rust-lang.org/stable/book/ch04-02-references-and-borrowing.html.

[13] 2025. Rust in the Android. https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html.

[14] 2025. Securing tomorrow's software: the need for memory safety standards. https://security.googleblog.com/2025/02/securing-tomorrows-software-need-for.html.

[15] 2025. Source Code of Soong. https://android.googlesource.com/platform/build/soong.

[16] 2025. Unsafe Rust - The Rust Programming Language. https://doc.rust-lang.org/stable/book/ch20-01-unsafe-rust.html.

[17] 2025. Validating References with Lifetimes - The Rust Programming Language. https://doc.rust-lang.org/stable/book/ch10-03-lifetime-syntax.html.

[18] 2025. What is Ownership? - The Rust Programming Language. https://doc.rust-lang.org/stable/book/ch04-01-what-is-ownership.html.

[19] 2026. AOSP. https://source.android.com.

[20] 2026. Cargo Toml. https://course.rs/cargo/reference/manifest.html.

[21] 2026. Drone OS. https://www.drone-os.com/.

[22] 2026. Nist handbook. https://www.nist.gov/programs-projects/nistsematech-engineering-statistics-handbook.

[23] 2026. Rust AIDL. https://developer.android.com/develop/background-work/services/aidl.

[24] 2026. Rust Bindgen. https://github.com/rust-lang/rust-bindgen.

[25] 2026. Rust Cxx. https://cxx.rs.

[26] 2026. Rust Fearless Concurrency. https://doc.rust-lang.org/book/ch16-00-concurrency.html.

[27] 2026. Rust FFI. https://doc.rust-lang.org/stable/core/ffi/index.html.

[28] 2026. Rust JNI. https://docs.rs/jni/latest/jni.

[29] 2026. Rust Safe and Unsafe. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html.

[30] 2026. Rust Unsafe Ability. https://doc.rust-lang.org/stable/nomicon/what-unsafe-does.html.

[31] 2026. Standard library developers Guide. https://std-dev-guide.rust-lang.org/policy/safety-comments.html.

[32] 2026. The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/rust/quick-start.html.

[33] 2026. Tock OS. https://tockos.org/.

[34] Hussain M. J. Almohri and David Evans. 2018. Fidelius charm: Isolating unsafe rust code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 248–255.

[35] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J Summers. 2020. How do programmers use unsafe rust? *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[36] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: finding memory safety bugs in rust at the ecosystem scale. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 84–99.

[37] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. 2017. System programming in rust: Beyond safety. In *Proceedings of the 16th workshop on hot topics in operating systems*. 156–161.

[38] William Bugden and Ayman Alahmar. 2022. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503* (2022).

[39] Andrey Bychkov and Vsevolod Nikolskiy. 2021. Rust language for supercomputing applications. In *Russian Supercomputing Days*. Springer, 391–403.

[40] Thomas Claburn. 2024. Rust developers at Google twice as productive as C++ teams.

[41] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A grounded conceptual model for ownership types in rust. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 1224–1252.

[42] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. 2021. Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis. *ACM Transactions on Software Engineering and Methodology* (2021).

[43] Mohan Cui, Shuran Sun, Hui Xu, and Yangfan Zhou. 2024. Is unsafe an Achilles' Heel? A Comprehensive Study of Safety Requirements in Unsafe Rust Programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[44] Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. Is rust used safely by software developers?. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 246–257.

[45] Shuanglong Kan, Zhe Chen, David Sanan, Shang-Wei Lin, and Yang Liu. 2018. An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing. *arXiv preprint arXiv:1804.07608* (2018).

[46] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. 21–26.

[47] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. 2024. An Empirical Study of {Rust-for-Linux}: The Success, Dissatisfaction, and Compromise. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 425–443.

[48] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John CS Lui. 2022. Detecting cross-language memory management issues in rust. In *European Symposium on Research in Computer Security*. 680–700.

[49] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing unsafe rust programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 234–245.

[50] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 841–856.

[51] Ian McCormack, Tomas Dougan, Sam Estep, Hanan Hibshi, Jonathan Aldrich, and Joshua Sunshine. 2024. A mixed-methods study on the implications of unsafe Rust for interoperation, encapsulation, and tooling. *arXiv preprint arXiv:2404.02230* (2024).

[52] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 245–258.

[53] Vikram Nitin, Anne Mulhern, Sanjay Arora, and Baishakhi Ray. 2024. Y uga: Automatically Detecting Lifetime Annotation Bugs in the Rust Language. *IEEE Transactions on Software Engineering* (2024).

[54] David J Pearce. 2021. A lightweight formalism for reference lifetimes and borrowing in Rust. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43, 1 (2021), 1–73.

[55] Boqin Qin, Yilun Chen, Haopeng Liu, Hua Zhang, Qiaoyan Wen, Linhai Song, and Yiying Zhang. 2024. Understanding and detecting real-world safety issues in Rust. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1306–1324.

[56] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 763–779.

[57] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *IEEE Symposium on Security and Privacy*. 1544–1561.

[58] Eric Reed. 2015. Patina: A formalization of the Rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02* 264 (2015).

[59] Ayushi Sharma, Shashank Sharma, Sai Ritvik Tanksalkar, Santiago Torres-Arias, and Aravind Machiry. 2024. Rust for embedded systems: current state and open problems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2296–2310.

[60] Sewen Thy, Andreea Costea, Kiran Gopinathan, and Ilya Sergey. 2023. Adventure of a lifetime: Extract method refactoring for rust. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 658–685.

[61] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R Lyu. 2021. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–25.

[62] Wei Xu, Daniel C DuVarney, and R Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*. 117–126.

[63] Wenzhang Yang, Linhai Song, and Yinxing Xue. 2024. Rust-lancet: Automated ownership-rule-violation fixing with behavior preservation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[64] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. 2023. Ownership guided C to Rust translation. In *International Conference on Computer Aided Verification*. Springer, 459–482.