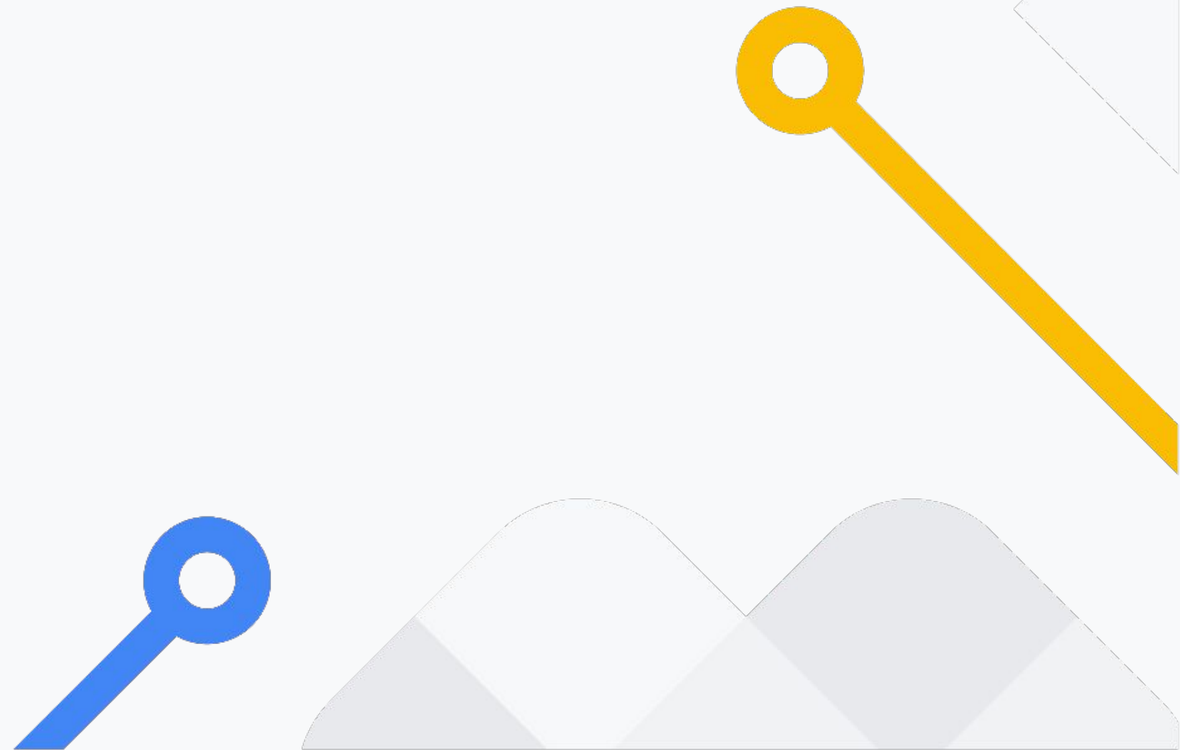
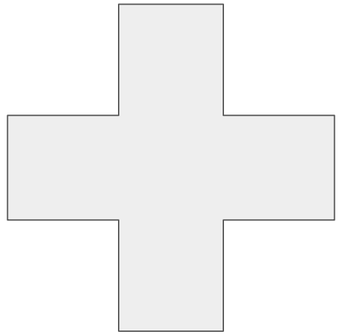
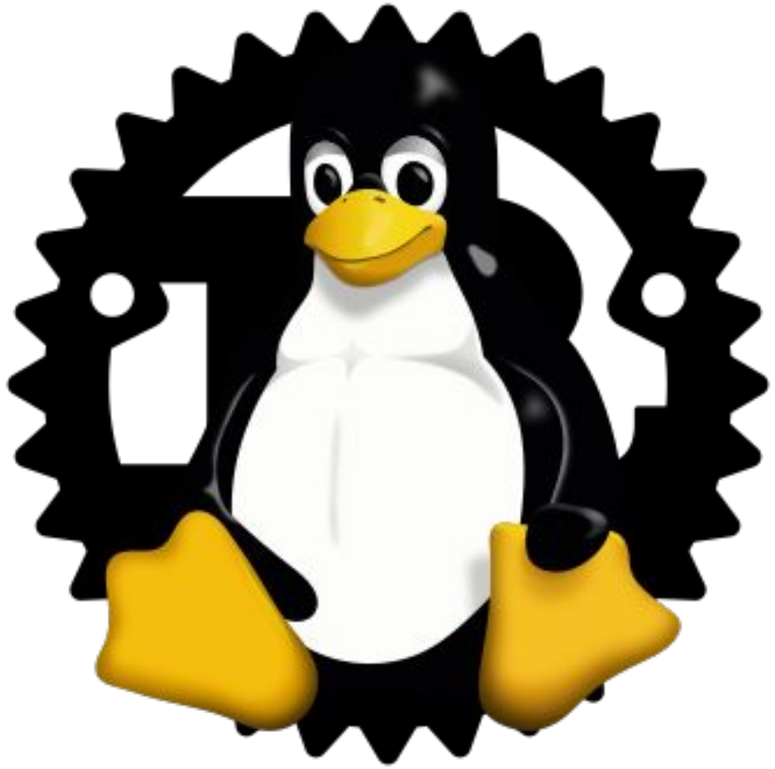
Transitioning to Rust

Jeff Vander Stoep - Android Team - Google

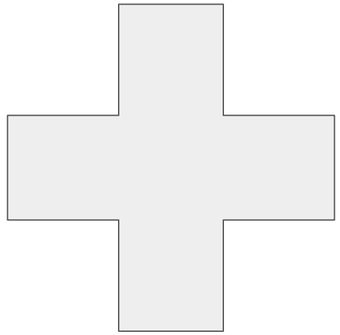


Rust in Android’s 6.12 kernel





Rust in Android’s C standard library



whoiam

- Member of the Android Security Team since 2014
  - **This is not a security talk!!!** (mostly)

- Software Engineer
- Avid Android user since 2010
  - Currently, happy Pixel 9 Fold user!

This is a talk about building better software, and through better software, better products.

“We who cut mere stones must always be envisioning cathedrals”
- Quarry worker’s creed

— Andrew Hunt, The Pragmatic Programmer

Android users (like myself!) care about

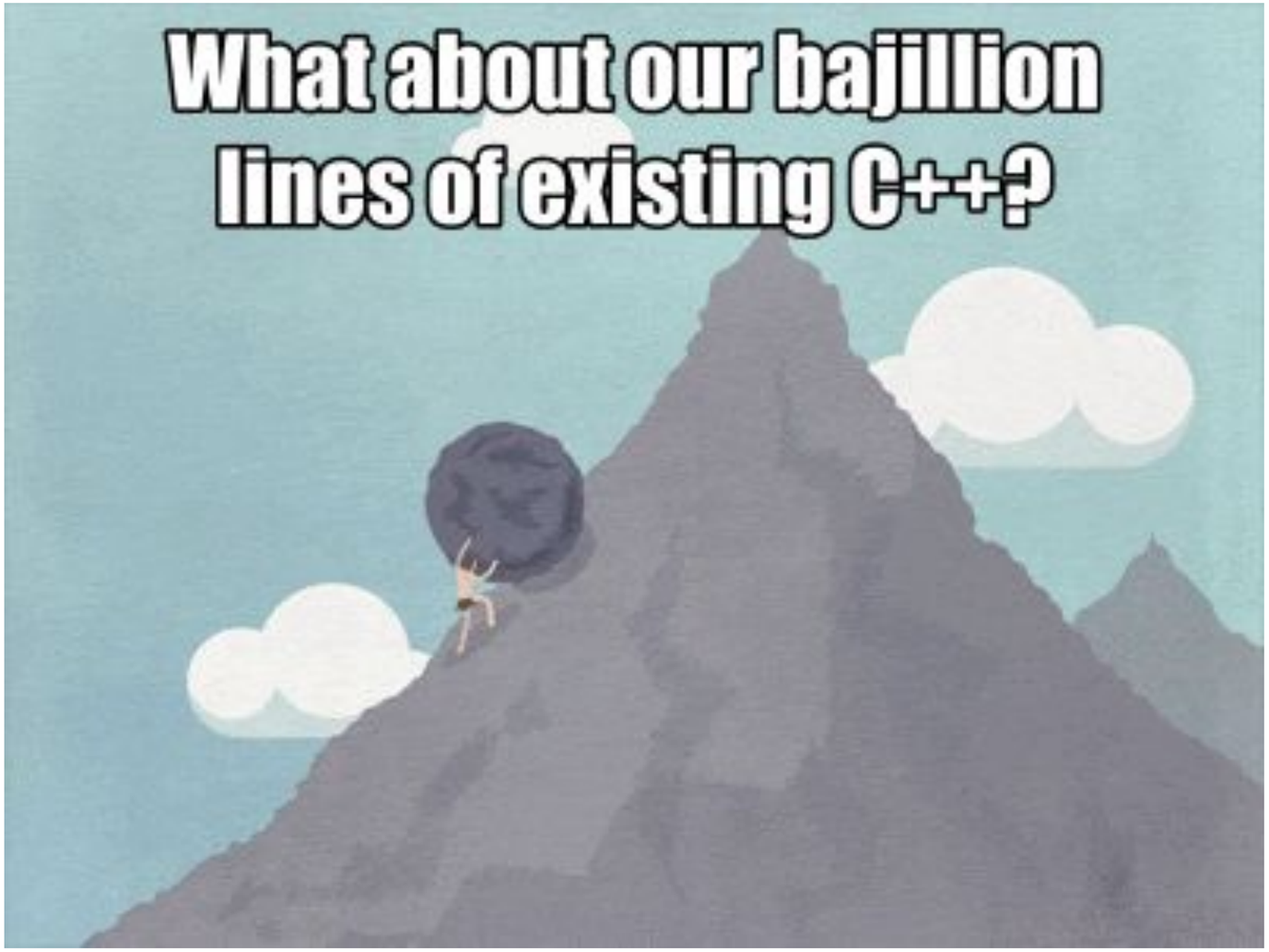
- Battery life
- Responsiveness
- Performance
- Stability
- Reliability
- Security
- Privacy
- ...

We regularly have to make tradeoffs between these things. Often due to our programming language choices.

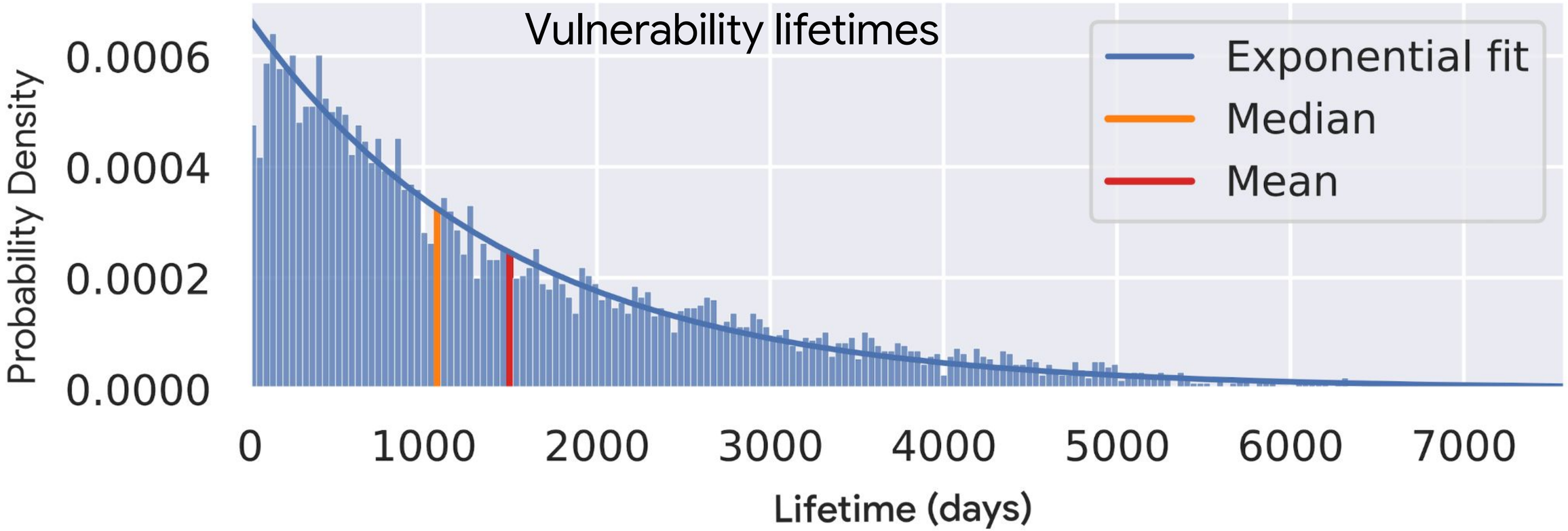
Can we make fewer tradeoffs with better tools?

Like Rust?

But...



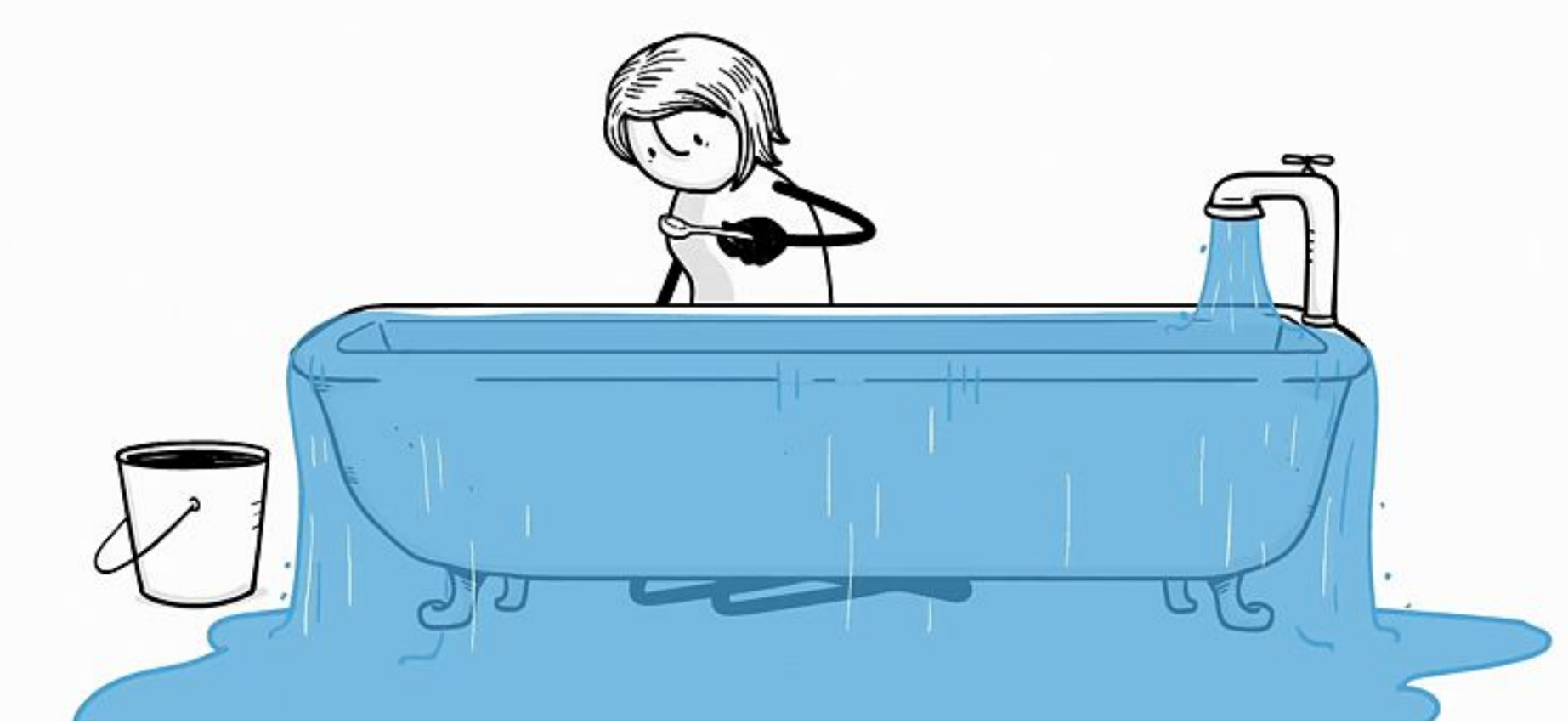
The data shows us where to focus



Not just vulnerabilities! Bug lifetimes follow an exponential distribution too.

Alexopoulos et al. "How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes". USENIX Security 22.

What’s the most important thing to focus on?



Rewrite all your old C++?

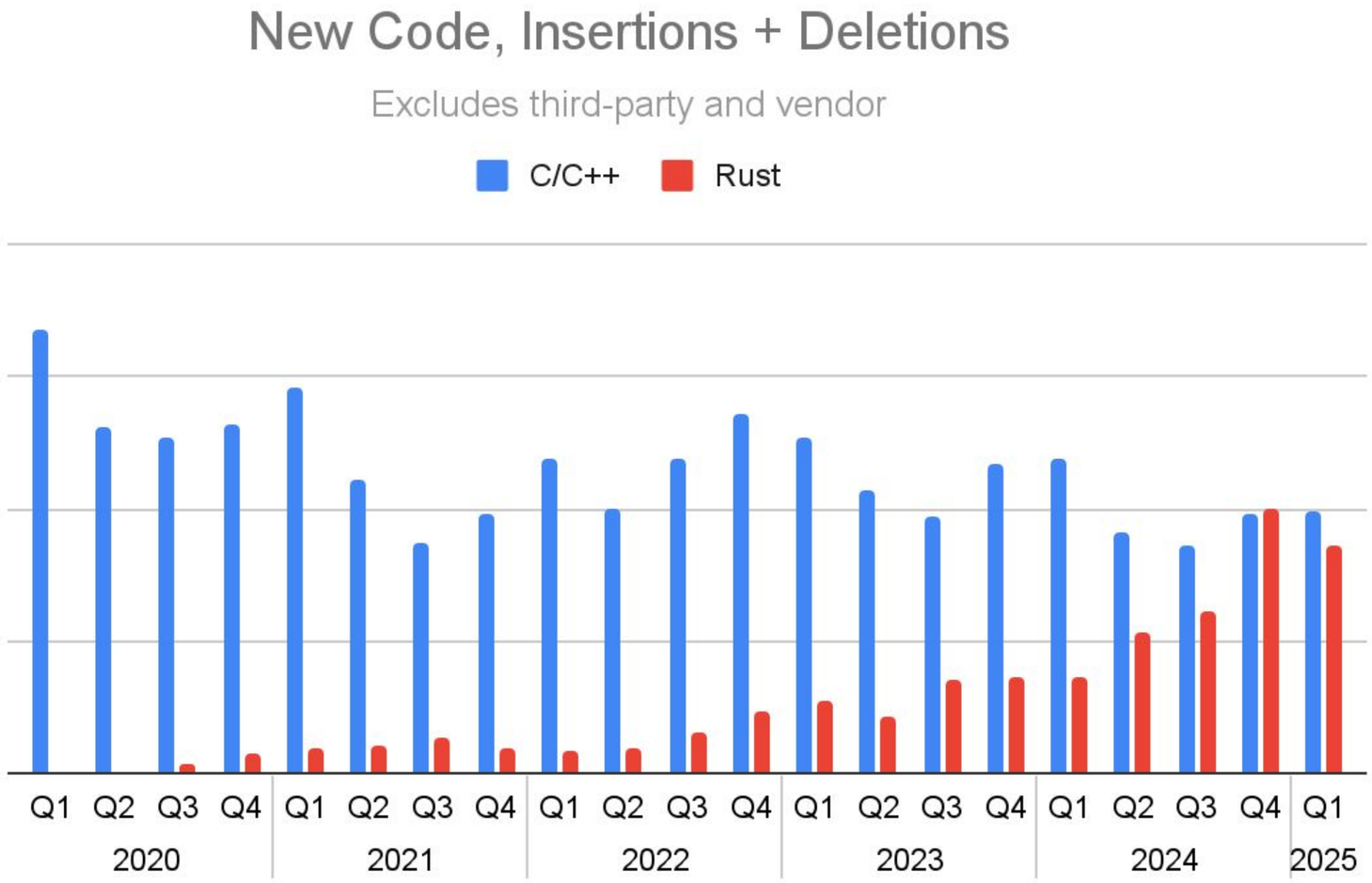
Nope!

The math is very different from other engineering problems. And that can be difficult to convey.

The riskiest code is the code that's just about to enter your codebase.

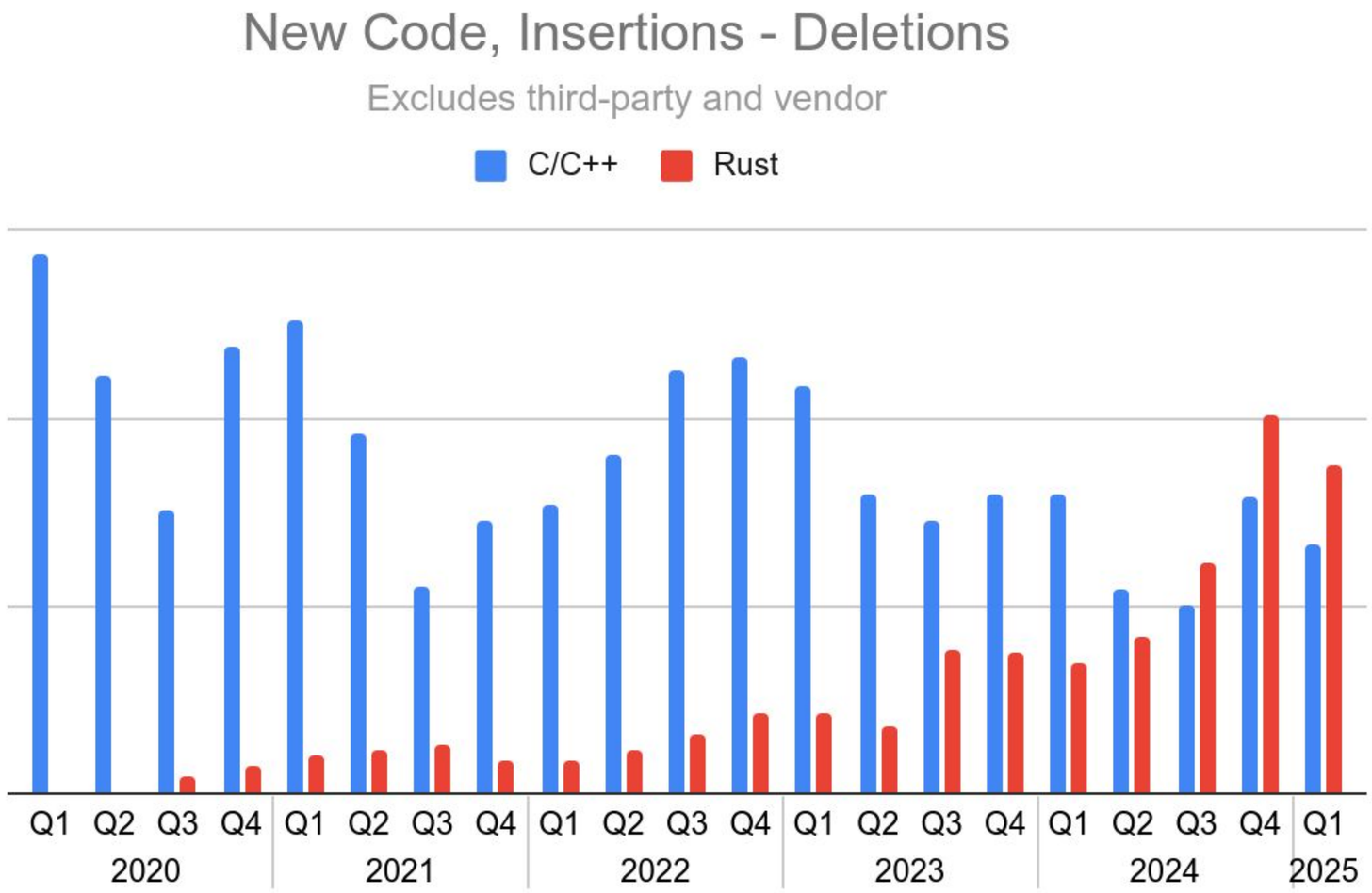
How can we fix that?
With better tools!

First party development in Android



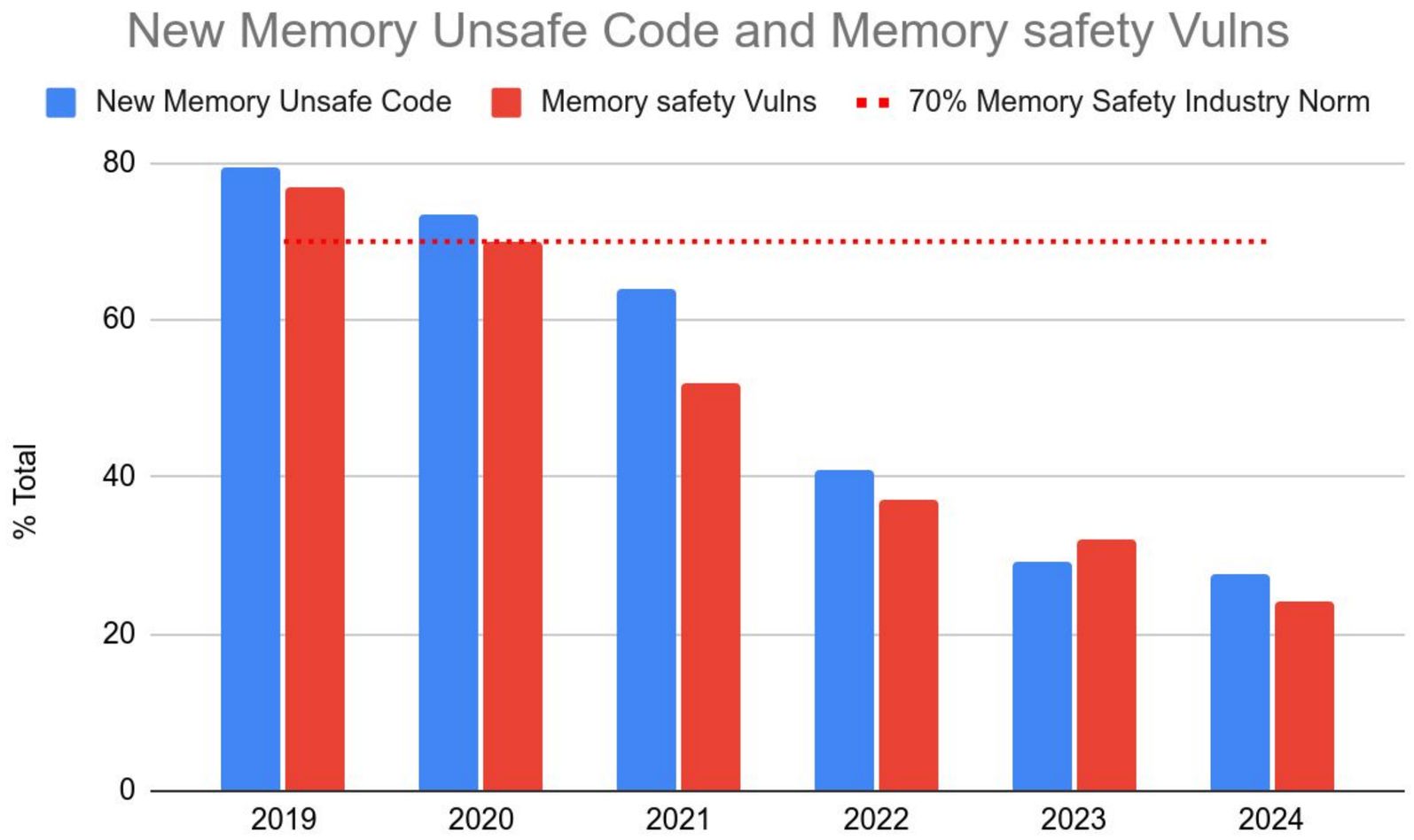
https://security.googleblog.com/2021/04/rust-in-android-platform.html

First party development in Android



https://security.googleblog.com/2021/04/rust-in-android-platform.html

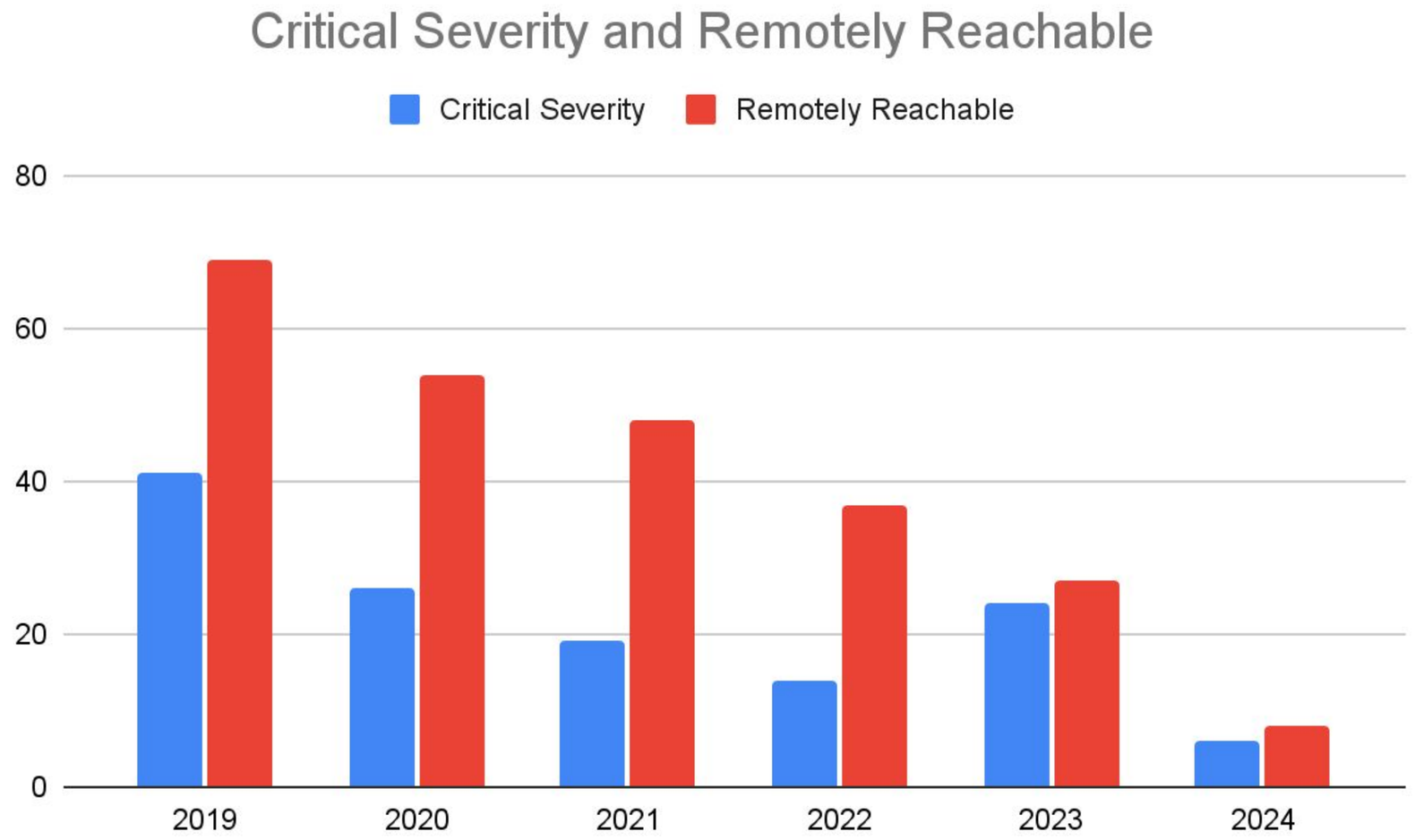
Vulnerability quantity decreases



https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

Vulnerability severity decreases



https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html

https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html

Security improved while security costs decreased

- Reduced sandboxing needs (memory, CPU savings).
- Fewer mitigations (Less CPU, reduced codesize).
- Less infra (reduced fuzzing needs).
- Less triage, remediation, patching.

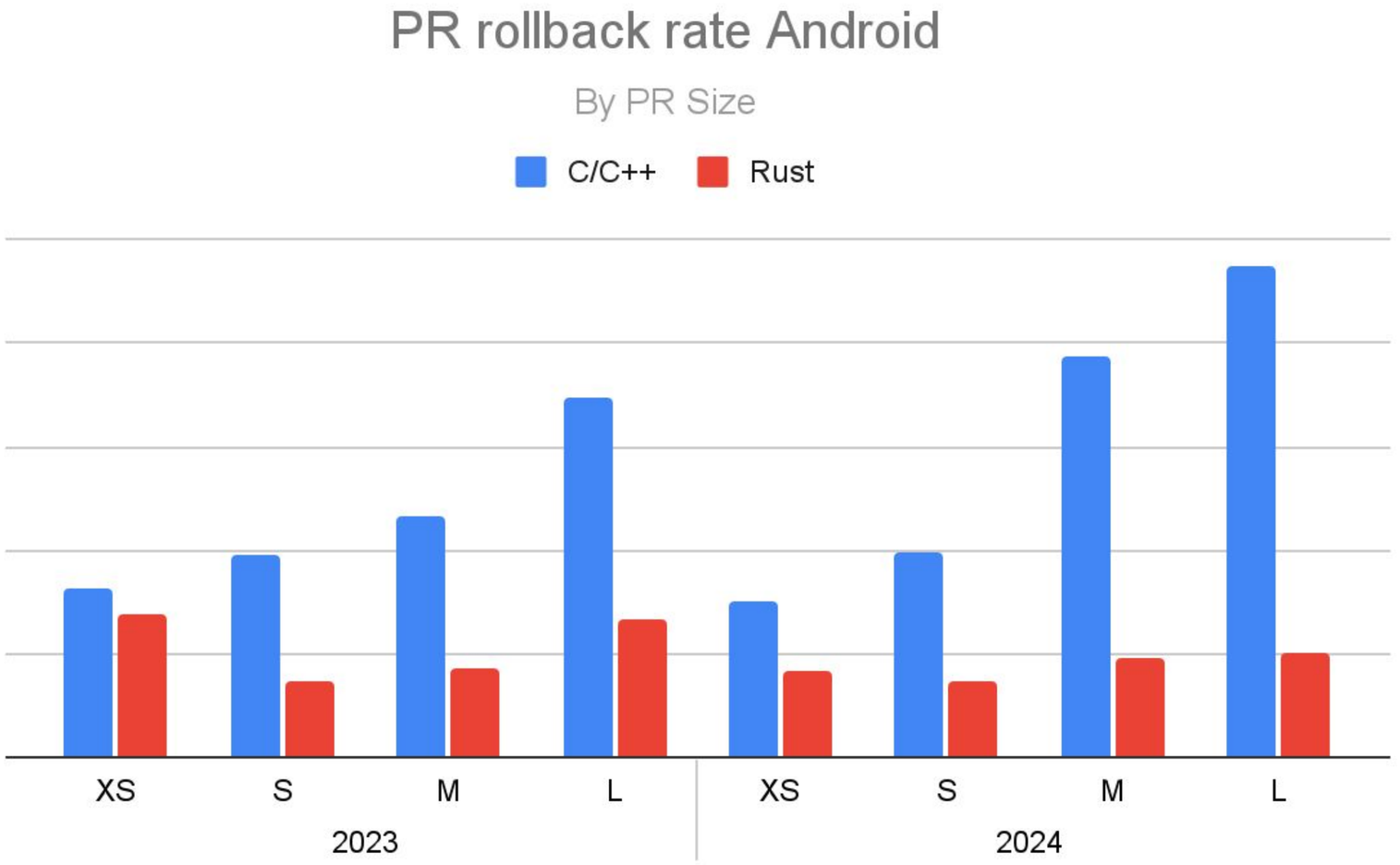
We're also seeing improvements in our key software delivery metrics.

https://dora.dev/guides/dora-metrics-four-keys/

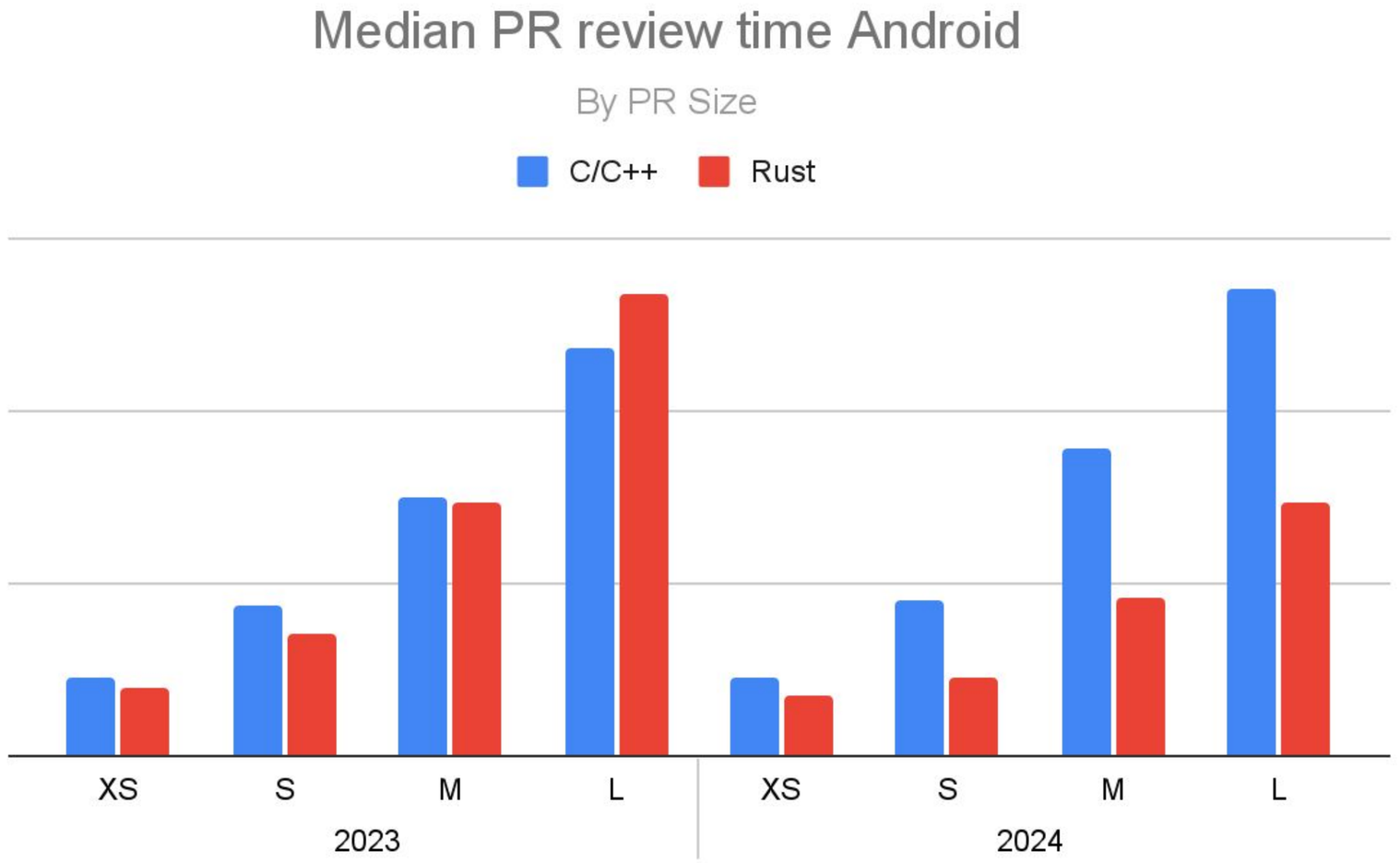
Throughput and stability

DORA's four keys can be divided into metrics that show the throughput of software changes, and metrics that show stability of software changes.

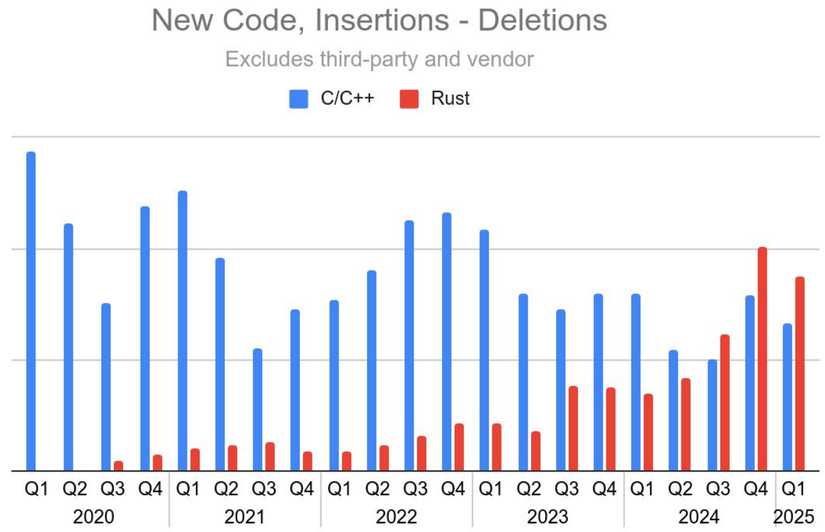
Change stability

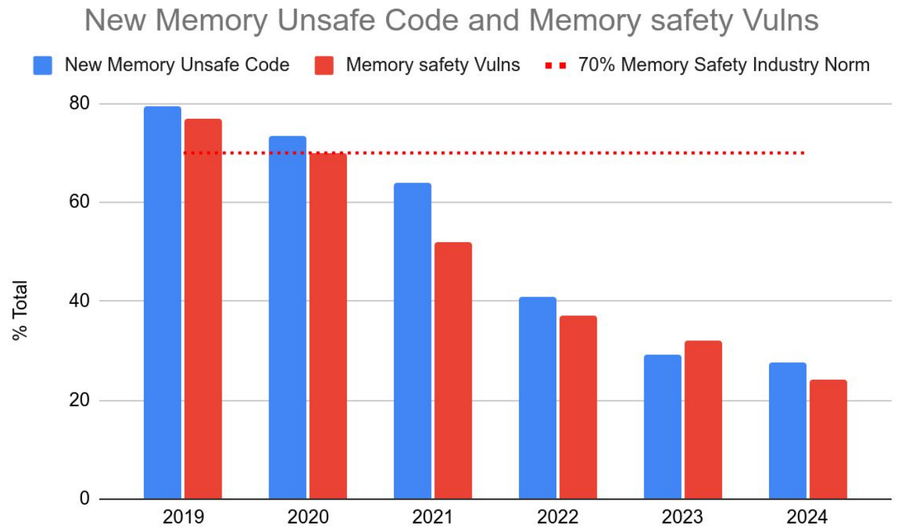


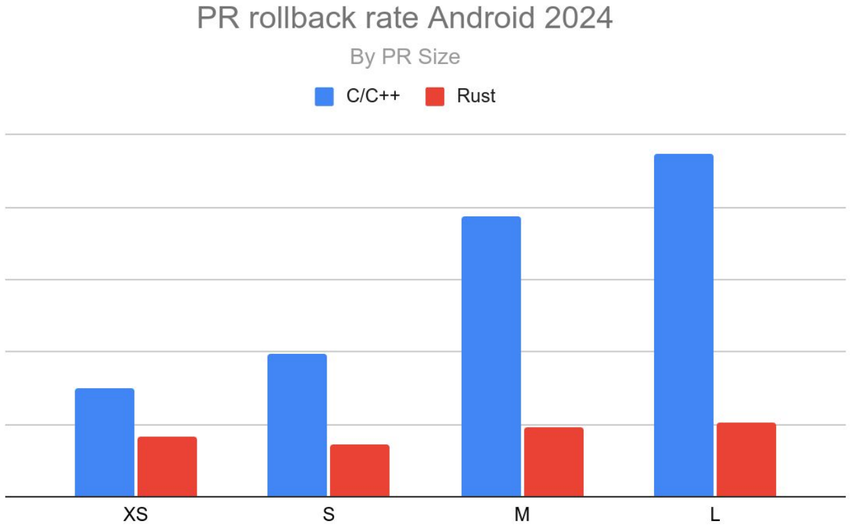
Change throughput

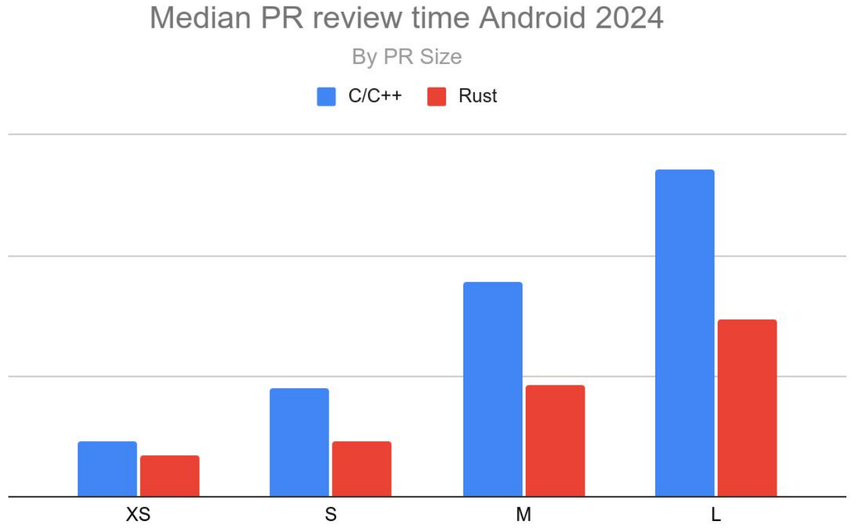


Better tools allow fewer tradeoffs!









Better security may...

Better security is achieved by spending more resources on it.

Better security is achieved while spending fewer resources on it.

Better security also improves non-security product goals, at lower cost.

But...introducing a new language is non-trivial.

How to do it successfully.

Android team’s game plan

- Stage 1: Initial support - Focus on core needs and build basic support for common use-cases.
- Stage 2: Pilots - Validate support and build confidence by successfully shipping pilot projects.
- Stage 3: Rust by exception - Control growth by approving new projects and expanding support.

Android team's game plan

- Stage 4: General availability - Remove restrictions as organic adoption meets demand.
- Stage 5: Rust recommended - Encourage wider adoption by addressing blockers for hesitant teams.
- Stage 6: (optional) C/C++ by exception - Make C and C++ the exception, guiding suitable projects to Rust.

But first! Stage 0: Customer-focused Rust team

- A key driver for Rust's adoption in Android has been a customer-focused Rust team whose focus shifts throughout the stages to meet evolving needs.
- The effectiveness of this team is not measured by what they build, but rather by the success of their customers in integrating and deploying Rust.
- This is who will be implementing the stages.

Stage 1: Initial support - laying the foundation

- Goal: Integrate basic Rust support for common project needs.
- Consider: Build integration, language interop, tools/tooling, bindings to frameworks and libraries, training.
- Exit criteria: Identify suitable pilot projects.
  - **Get out of this stage as quickly as possible!** Don’t get hung up on corner cases or niche use cases.

https://security.googleblog.com/2021/06/rustc-interop-in-android-platform.html

https://security.googleblog.com/2021/05/integrating-rust-into-android-open.html

Stage 2: Pilots - Build Confidence Through Success

- Goal: Validate the support, address gaps.
  - **The pilots themselves need to be successful**! Ship code!
- Consider: Having more than one pilot. Choosing low risk projects.
- **Align with the customer team that validating Rust support is a key deliverable for them!**
  - Sure way to fail? Neglect providing expert support to pilot projects.
- Exit criteria: Pilots have successfully shipped.

Stage 3: Rust by exception - controlled expansion

- Goal: Grow carefully, ensuring customer teams succeed.
- Consider: A project approval process that considers project needs and Rust team capacity in selection process.
- Exit criteria: Project proposals are being routinely approved, indicating a maturing language support, and sufficient support capacity. Announce general availability.

Stage 4: Rust General Availability

- Goal: Ensure supply meets demand.
- Consider: Scaling support via community - support chat rooms, expert reviewers.
- Exit criteria: Organic demand is met. Teams naturally inclined to adopt Rust have done so.

Stage 5: Rust Recommended - Generating Demand

- Goal: Generate additional demand by convincing more teams to use Rust for new features and projects.
- Consider: Active outreach to understand hesitancy or blockers.
- Exit criteria: most blockers for most teams are addressed. There is an understanding of where C or C++ remain the right choice.

Stage 6: (Optional) C/C++ by Exception

- Goal: Rust is the default choice. A process, policy, incentive, or nudge exists to move the default when reasonable. Maximize for long term gains, not short term losses.
- Consider: **Requires organizational alignment that transitioning languages is an objective**. Document automatic exceptions and have straightforward mechanism for exceptions and/or area overrides.
- Exit criteria: Never. The goal is not to stop using C/C++ entirely but rather to establish Rust as the default.

Where next?

- Follow these same stages in new areas.
  - Rust-for-Linux, Rust in firmware.
- Expand the definition of “new code” through better interop.
- Warning, these can be shiny traps.
  - Focus on the basics first.
  - Focus on customers first.
  - Focus on deployment first.
  - The 80/20 rule applies here.