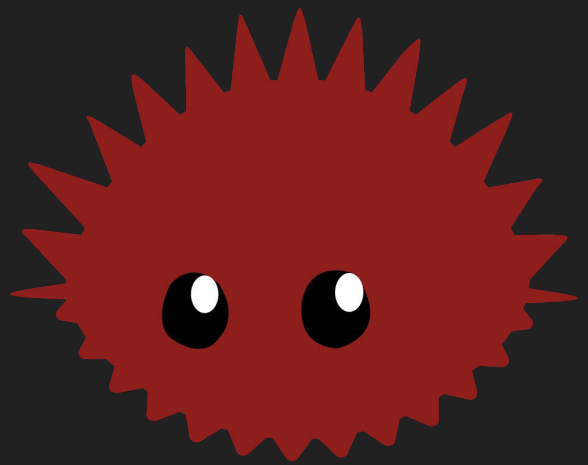
Safe && Portable Data Structure Design

(in Rust)

Code and Supply Lighting Talk, Dec 2021

Memory Safety: How are we doing?

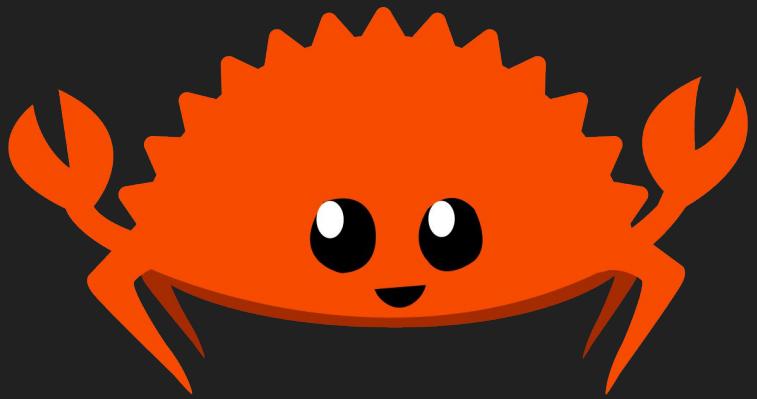


The Immortal Vulnerability Class

- Memory safety: exploit might mean turing-complete control of target process!
  - E.g. ROP chain for a heap buffer overflow
- 2012: research surveyed **30 years** of failed C/C++ memory protections
  - “SoK: Eternal War in Memory” by Szekeres et. al. (2012) *[1]*
- 2021: "Out-of-bounds Write" is #1 vulnerability of the year
  - Per the MITRE CWE Top 25 Most Dangerous Software Weaknesses *[2]*

[1] https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf
[2] https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html

Memory Safety:
We can do better!



About This Talk

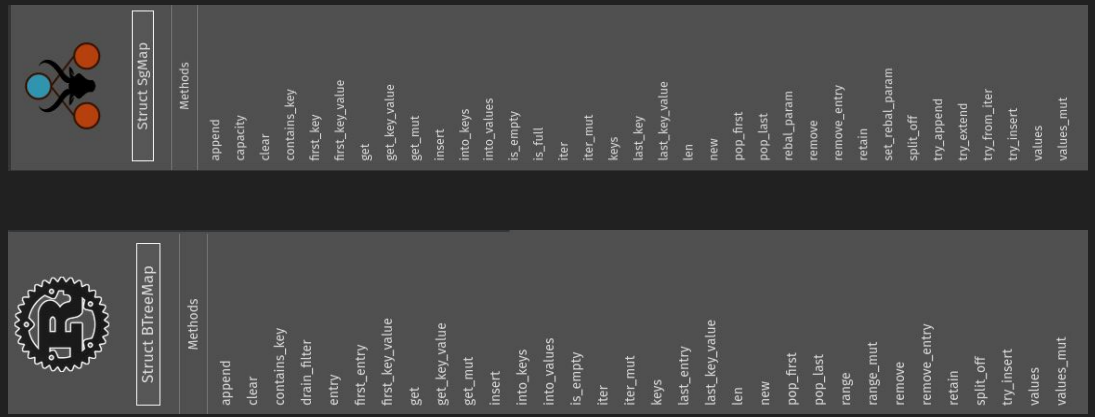
- Designing a Rust data structure library…
- ...that can run, w/o an OS, on any embedded device Rust targets…
  - E.g. **#![no_std]**
- ...and in which there is *provable** absence of memory corruption bugs
  - E.g. **#![forbid(unsafe_code)]**
  - *“Computer Scientist proves safety claims of the programming language Rust” **[1]***

* == Barring soundness bug in the compiler or an unsafe DLL/core::function, etc. ***No absolute security!***

[1] https://www.eurekalert.org/news-releases/610682

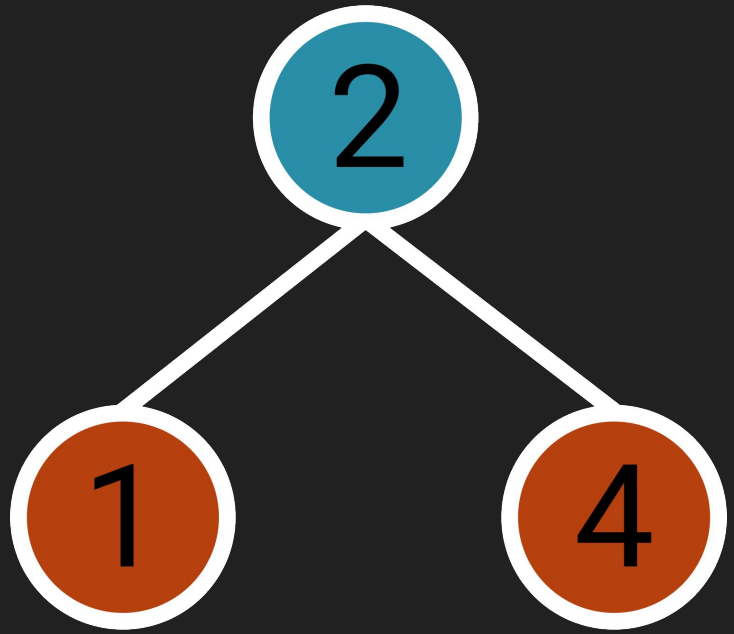
Alternative to **std::collections::{BTreeMap, BTreeSet}**

- https://github.com/tnballo/scapegoat, MIT Licence OSS
- **BTreeMap** has 32 APIs on nightly
- **SgMap** has 27 of those 32 on stable + 4 fallible API variants + 4 misc



How do ordered sets/maps work “under-the-hood”?

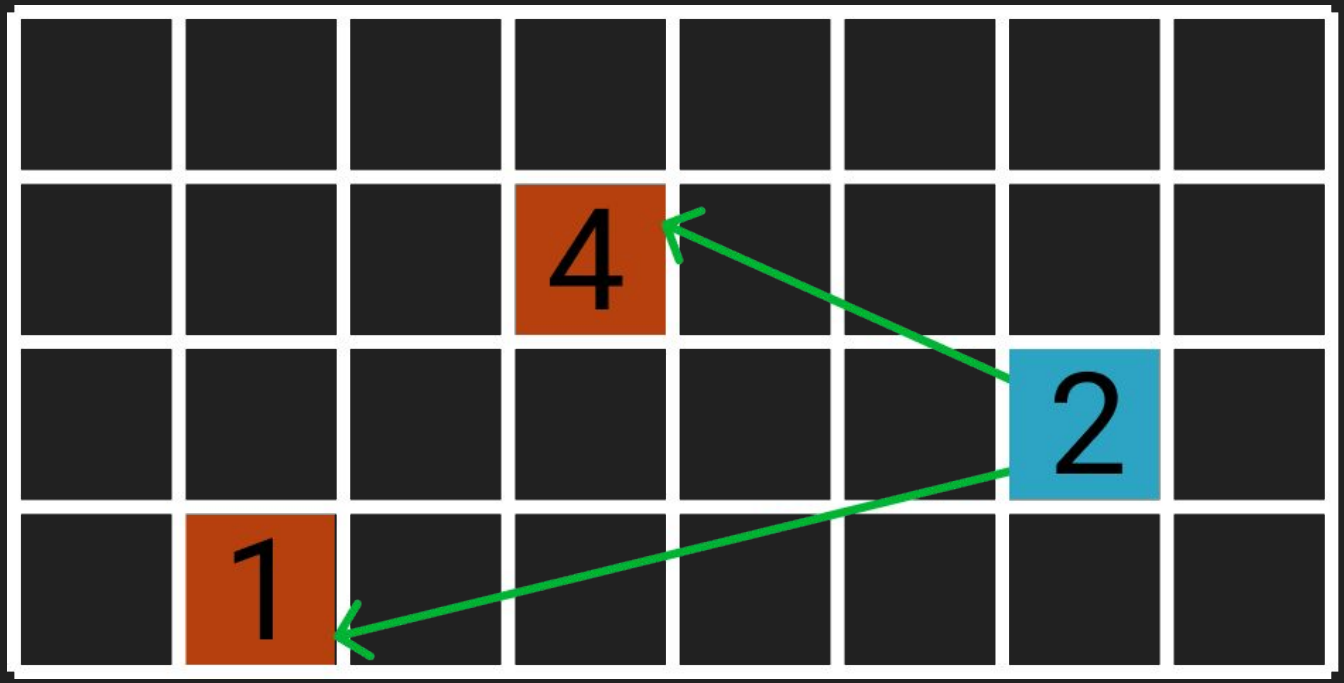
- Typically* implemented with a self-balancing search tree
  - Rust std: BTree
  - C++ STL: Red/Black Tree
  - Java.util: Red/Black Tree
- Retrieval is usually *O(log n)*
- Running example binary tree on right



* == CPython’s OrderedDict is a link list?!

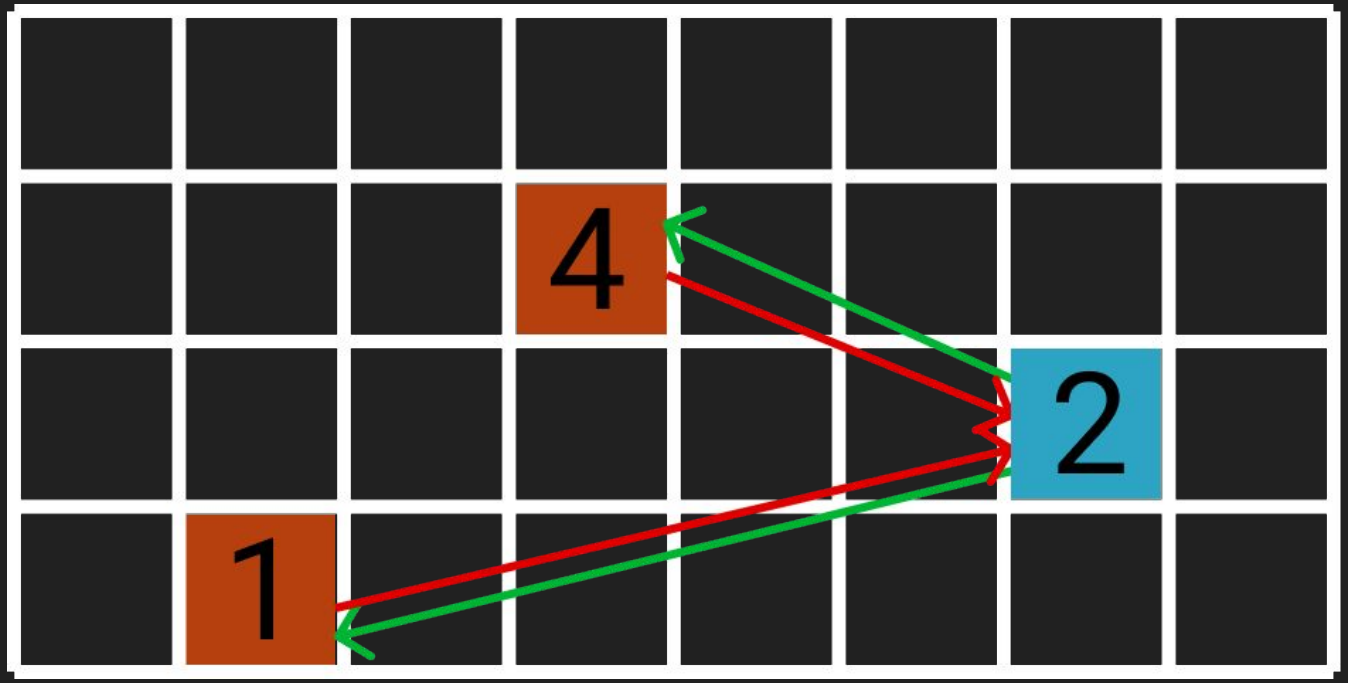
The Problem with Graphs/Trees in Rust

- On the heap, our example tree looks like this (owning references in green):



The Problem with Graphs/Trees in Rust

- If our algorithm also requires parent references (red), we have a problem!



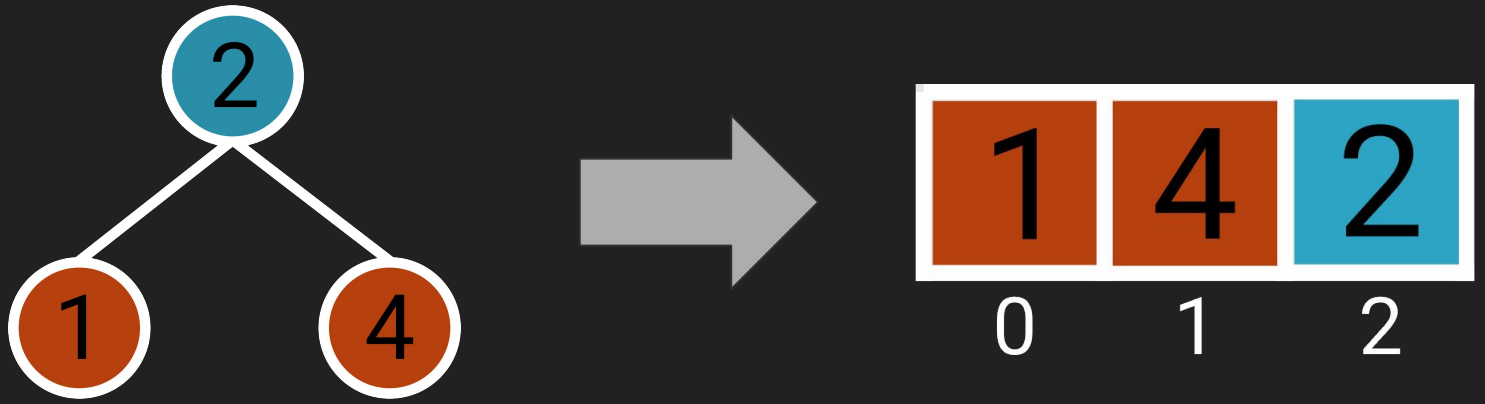
3 Solutions: the Good, the Bad, and the Ugly

- Bad: C-style raw pointers (**unsafe** keyword)
  - Can read/write arbitrary memory, all bets are off!
- Ugly: **Rc<RefCell<T>>** (interior mutability with smart pointers)
  - Must take a runtime check penalty, it's lipstick over **UnsafeCell**.
- Good: arena allocation (next slide)
  - May still need runtime checks, but portability is unlocked and code is safe/clean/maintainable.
  - Arena allocation is not unique to Rust, but was demonstrated in Rust as early as 2015 *[1]*

[1] http://smallcultfollowing.com/babysteps/blog/2015/04/06/modeling-graphs-in-rust-using-vector-indices

The Safe Solution: Arena Allocation

- Store tree as a vector of elements, use indexes instead of pointers:



- Scanning this storage left to right, we have a logical tree:
  - Leaf 1: no children, parent at index 2
  - Leaf 4: no children, parent at index 2
  - Root 2: left child at index 0, right child at index 1

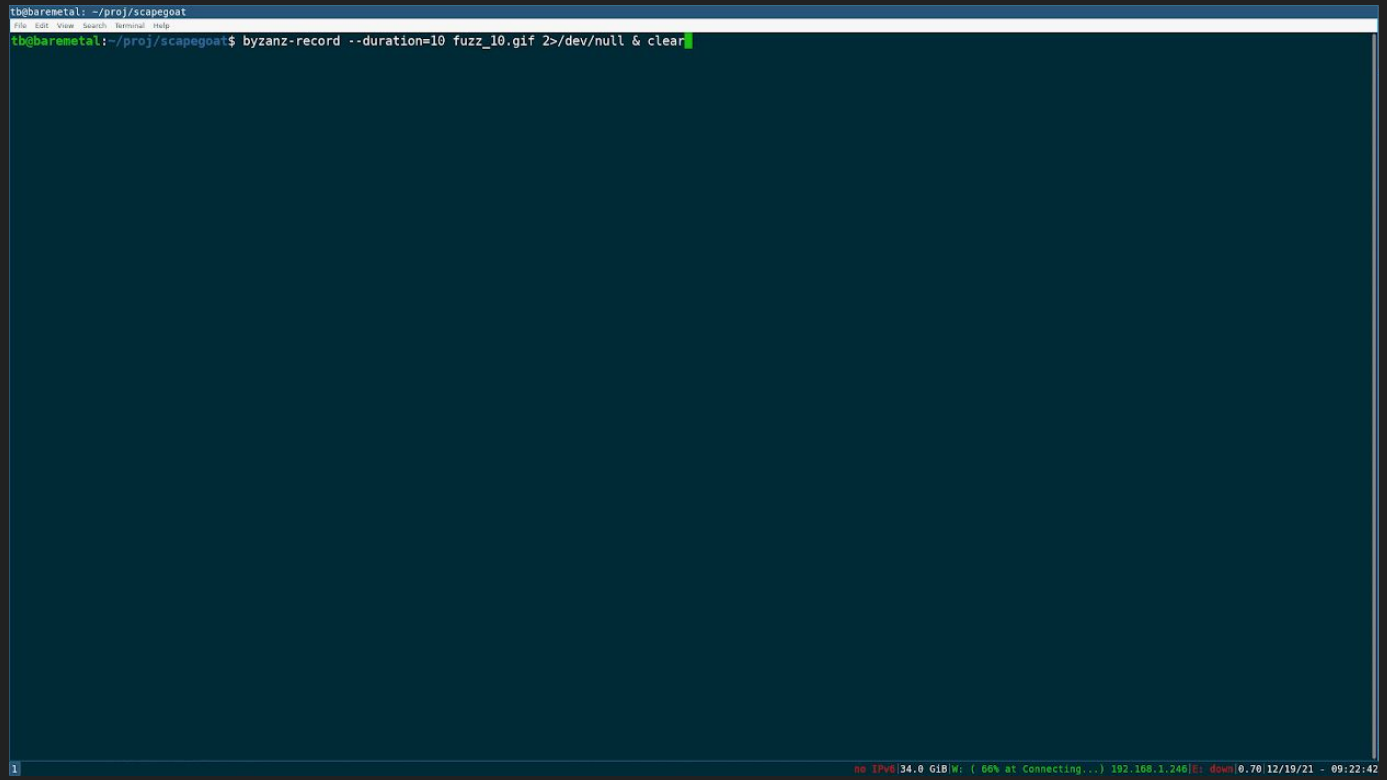
The Safe Solution: Arena Allocation

- **Portability:** know max capacity? Pre-allocate on the stack, no heap use!
  - Use fixed-size stack array instead of vector
- **Safety:** no raw pointers, no interior mutability checks
  - Index-based accesses may still be bounds-checked at runtime

Ok, it’s safe and portable. But is it robust and correct?

- **Robustness:** We can’t be sure!
  - OOB arena access means termination in Rust (e.g. “panic”)
  - But not memory corruption, like C or C++
- **Correctness:** arena doesn’t guarantee set/map logic is correct.
- “Differential fuzzing” can validate reliability and logical equivalence. Idea:
  - Use standard library’s set/map as a “known good” model, fuzz against it
  - Stress test all APIs, in random order and with random parameters, and check “lock step”

Validation: Differential Fuzzing with LLVM's LibFuzzer



https://github.com/rust-fuzz/cargo-fuzz

Takeaway

- Portability, safety, speed - with a little reframing, you can have all 3:
  - **Portable:** Stack-only mutable structures possible, no heap or garbage collection.
  - **Safe:** 1st-party static analysis provably eliminates vicious C/C++ bug classes.
  - **Fast:** Rust's speed is comparable to C/C++, often within single-digit percentage.

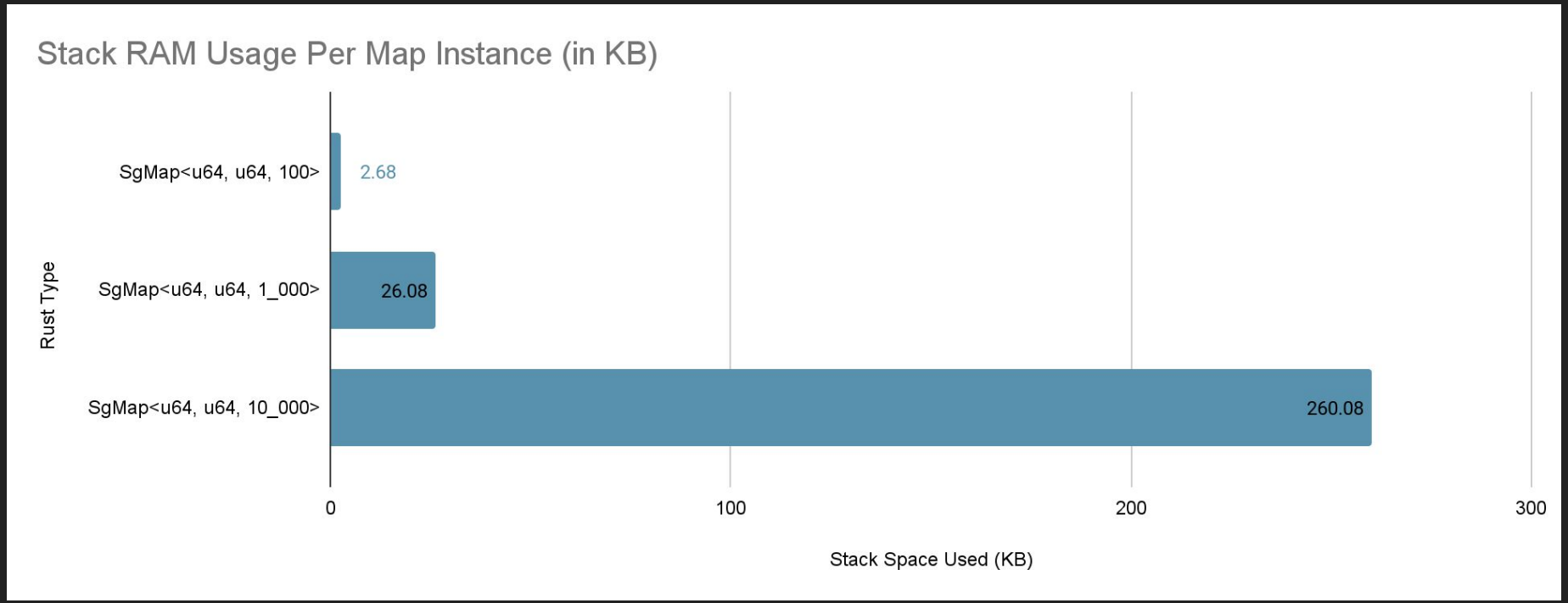
- Safe Rust is a limited kind of "formal verification":
  - **Limited:** Prove only one, domain-agnostic property - memory safety.
  - **But Practical:** Development speed suitable for many commercial businesses.
  - **Fuzzing** is supplemental, stochastic validation for properties the compiler can't prove.

Thank you!

- Crate: https://crates.io/crates/scapegoat
- Blog: https://tiemoko.com/blog/
- GitHub/Twitter: @tnballo

Extra Slides

Per-instance Map Size with Const Generics (Rust 1.51+)



What happened in that terminal scroll?

- Basic block level coverage-guidance
  - Inputs generated to maximize code coverage in programs under test
- Structure-aware mutation
  - Both API calls and their arguments were generated precisely for the program
- Differential comparison
  - Looking for high level logic bugs the specification of an “ordered set”, not just crashes