Hardening QEMU with Self-Correcting Fuzzing Pipelines

An AI-Driven Approach to Scaling Security Testing

Navid Emamdoost
Google Distributed Cloud Security Team

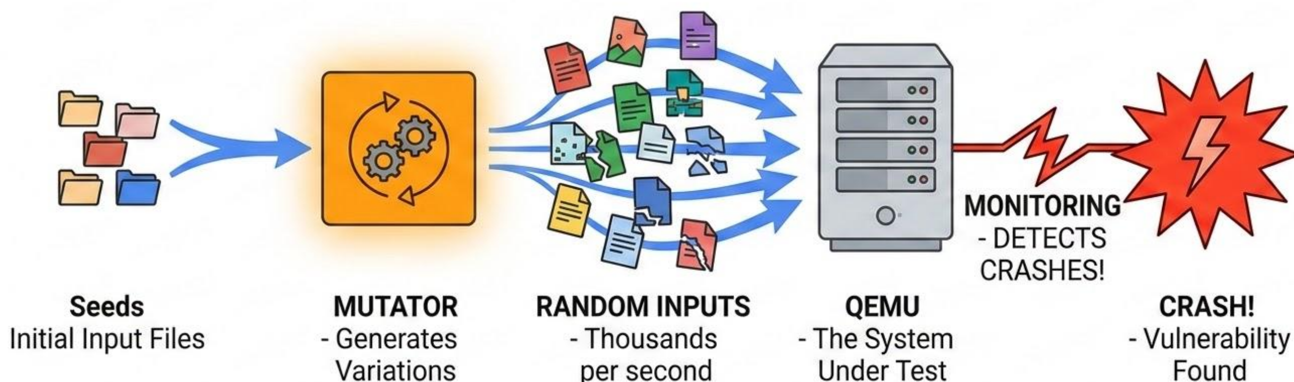
Open Source Summit - 2026

The Challenge

Why Fuzzing QEMU is Hard

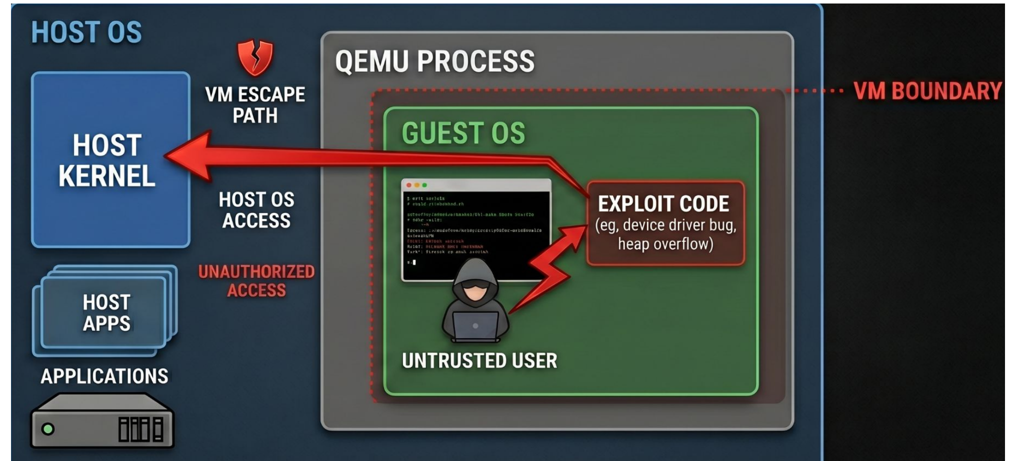
What is Fuzzing? (A Quick Refresher)

- An automated software testing technique that feeds a program invalid, unexpected, or random data.
- Goal: Find bugs, memory safety violations, and security vulnerabilities by triggering unexpected states.
- A "fuzzer" (like libFuzzer) systematically mutates inputs and monitors the target for crashes.



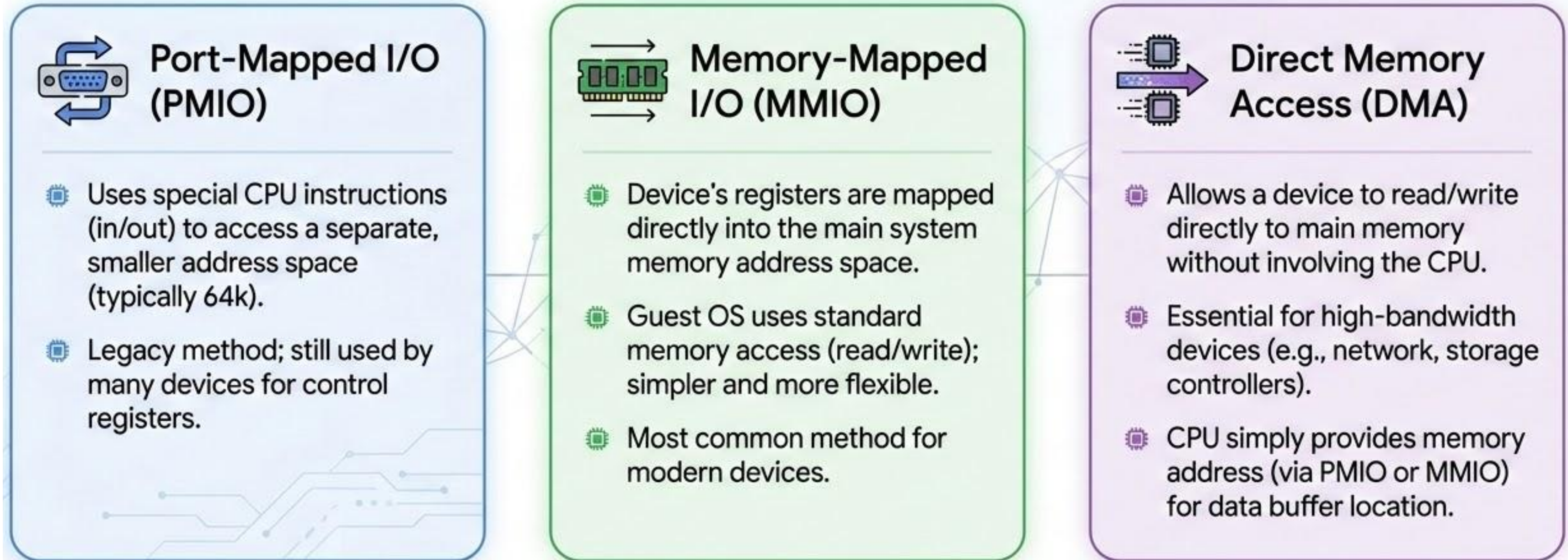
Why Fuzz QEMU? The Critical Attack Surface

- QEMU is the hypervisor, emulating over 200 unique hardware devices (USB, network, storage).
- It forms the ultimate security boundary between a guest OS and the host.
- The Threat: A bug in any device emulation can lead to a VM escape, compromising the entire host system.



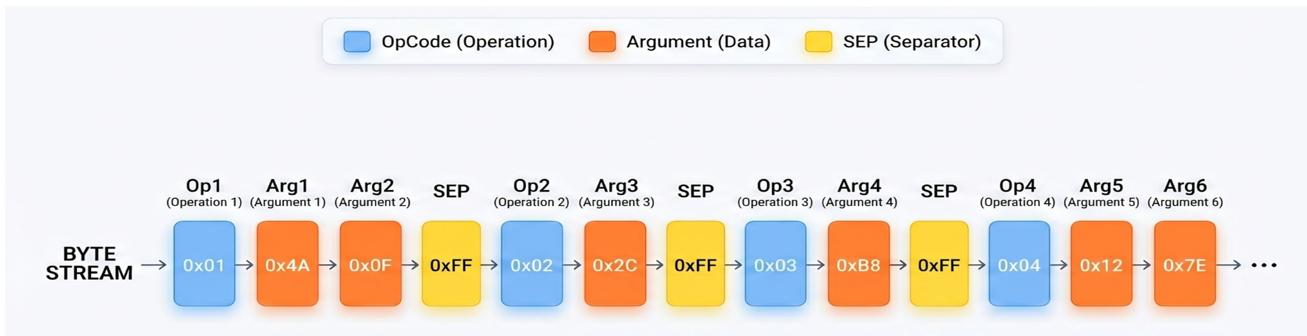
Google

The Three Channels of Device Communication



How the Generic Fuzzer Works: From Bytes to Commands

- The fuzzer translates a raw byte stream into a sequence of I/O operations using a SEPARATOR token.
- Why a separator?
  - Variable-Length Commands: An op_write can be long, op_reset can be short.
  - Mutation Stability: Deleting one byte only affects one command, not the whole stream.
- Parsing Process:
  - Scan for Separator -> Determine Opcode -> Extract Arguments -> Execute



Google

The Generic Fuzzer's "Language"

- The fuzzer interprets the input as a sequence of simple commands—an agnostic "language" for talking to any device.
- Core Operations:
  - In [Index, Offset] - Read from Port I/O
  - Out [Index, Offset], Value - Write to Port I/O
  - Read [Index, Offset] - Read from MMIO / RAM
  - Write [Index, Offset], Value - Write to MMIO / RAM
  - DMA buf[] - Populate DMA reads with a pattern
- Access size (1, 2, 4, 8 bytes) is encoded in the opcode.

Current State of Generic Fuzzers

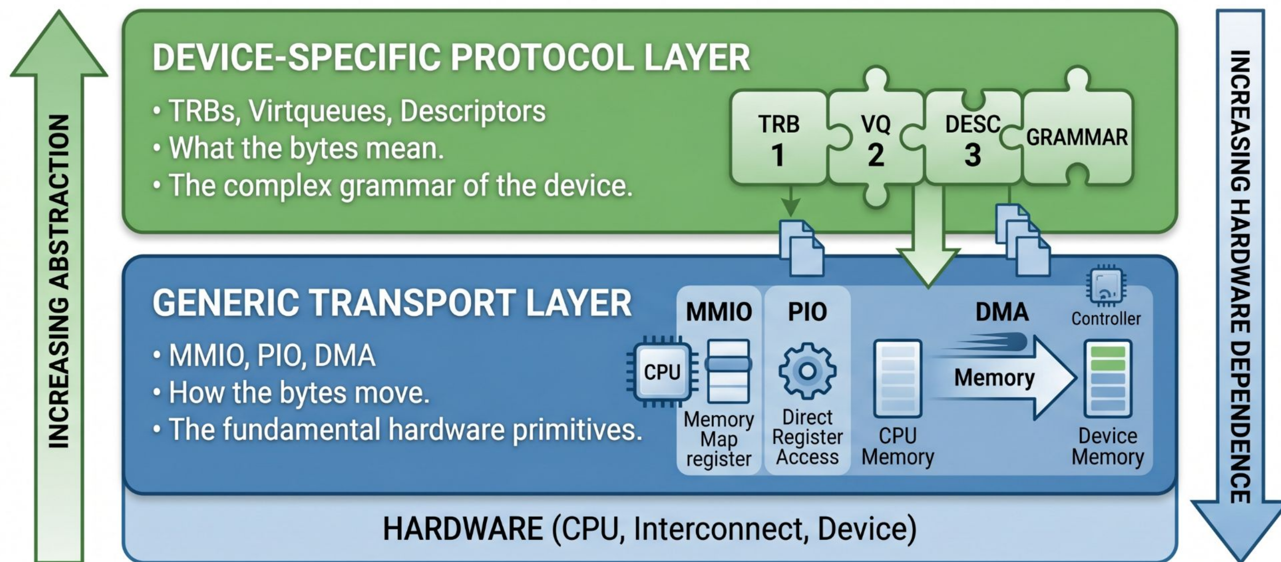
- QEMU is integrated in OSS-Fuzz – credit to Alexander Bulekov
  - Continuous fuzzing for high-profile open source projects
- Currently 18 generic fuzz targets
- At Google, we support a set of devices
  - Majority were not being fuzzed
- Added 50 generic fuzz targets
- Increased per-device fuzz coverage from 26.22% to 56.50%
- Still high-performance devices are showing lower coverage

The Core Challenge: Transport vs. Protocol

The Generic Fuzzer's Blind Spot

The Two Layers of Device Emulation

- Device communication isn't monolithic; it's a layered stack
- The Generic Fuzzer operates almost entirely at the bottom layer



Protocol is Everywhere: A Look Across Devices

- This layered model applies to virtually all modern devices
- The "real" work happens at the protocol layer, which is unique to each device

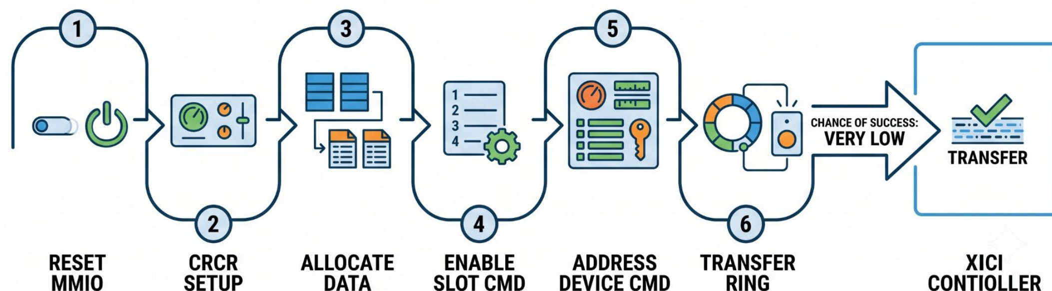
Device Class	Generic Transport	Device-Specific Protocol
xHCI USB	MMIO + DMA	TRBs, command/event rings
e1000 NIC	MMIO + DMA	TX/RX descriptor rings
virtio-net	PCI/MMIO	Virtqueues, virtio-net headers
NVMe	MMIO + DMA	Submission/Completion queues

The Semantic Gap

- Modern virtual devices are not just memory buffers; they are highly intricate state machines.
- A fuzzer mutating random bytes continuously hits shallow parser errors (bad checksums, invalid IDs).
- Without understanding the protocol's structure, the fuzzer gets permanently blocked at the "front door".
- Deep, critical code paths remain completely unreachable by simple, generic mutation.

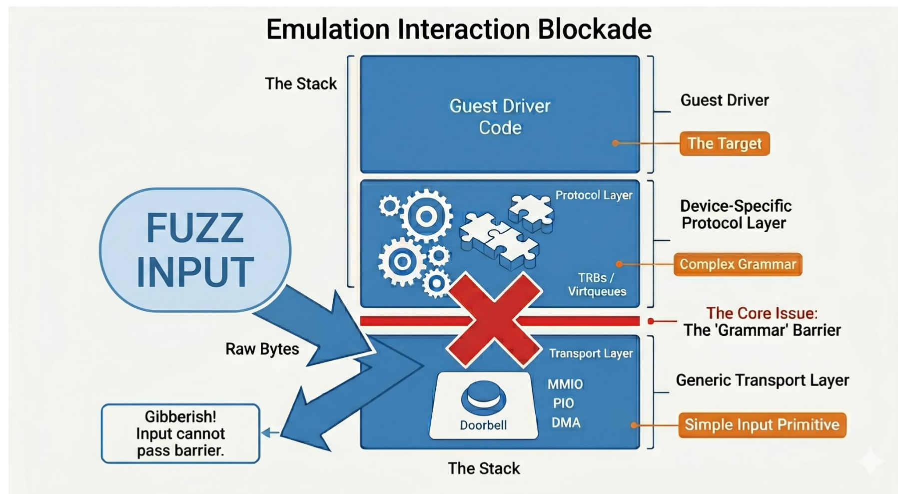
Example: The XHCI State Machine Challenge

- To get an XHCI controller to process a transfer, the fuzzer must accidentally guess how to:
  - Write specific magic values to MMIO registers to reset the controller.
  - Set up a valid Command Ring Control Register (CRCR).
  - Allocate and format a Device Context Base Address Array.
  - Issue an "Enable Slot" command and wait for the completion event.
  - Issue an "Address Device" command.
  - Finally: build a perfectly formatted Transfer Ring in memory and ring the doorbell.



The "Grammar" Problem & Chance of Success

- **The "Grammar" Problem:** Stateful protocols expect a strict sequence and format of commands.
- **0% Success Rate:** Random mutation gets trapped in early error handlers; deep code remains unreachable.
- **The Solution:** Protocol-aware fuzzing using structured grammars to speak the device's native language.



The AI Solution

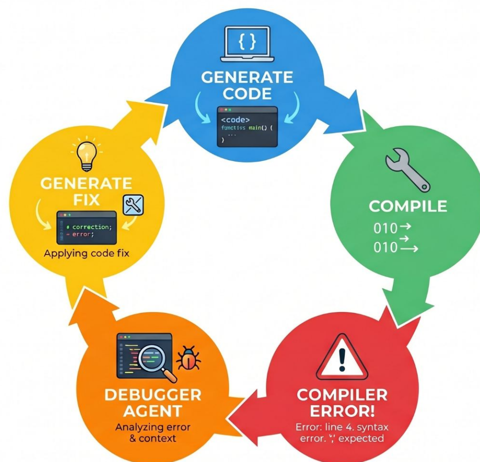
Teaching the Fuzzer to Speak "Protocol"

The Core Idea: A Dual-Agent AI Pipeline

- We separated the problem into two distinct roles:
  a. The Protocol Expert: Understands the "what" and "why" of the device protocol
  b. The Fuzz Writer: Handles the "how" of implementing the fuzzer

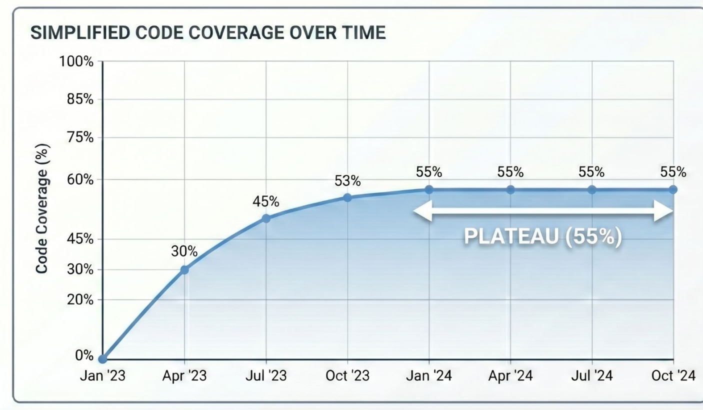
The Inner Loop: Self-Correcting Code Generation

- LLMs rarely generate perfect code on the first try
- Our automation pipeline captures compilation errors and feeds them back into a specialized Debugger Agent
- This loop runs automatically until the code compiles successfully



The Outer Loop: Overcoming Strategic Blockers

- What if the code compiles but isn't effective? This is the fuzzing plateau
- We use a second, "outer loop" for strategy
- We feed coverage reports back to the Protocol Expert
- Prompt: "Analyze this coverage data. Identify why we are stuck and propose a new strategy to get past this blocker."



Case Study: Escaping the XHCI HALT State

- The Block: Our XHCI fuzzer would put endpoints into a HALT state and get stuck
- Diagnosis: The Protocol Expert identified the HALT state from the source and spec as a terminal condition
- Solution: It proposed a new "sanity check" for the mutator. If the endpoint is halted, first inject a Reset Endpoint command
- The Result: This simple change immediately broke the plateau and led directly to discovering a new heap-buffer-overflow

The New Language: Protocol-Aware Opcodes

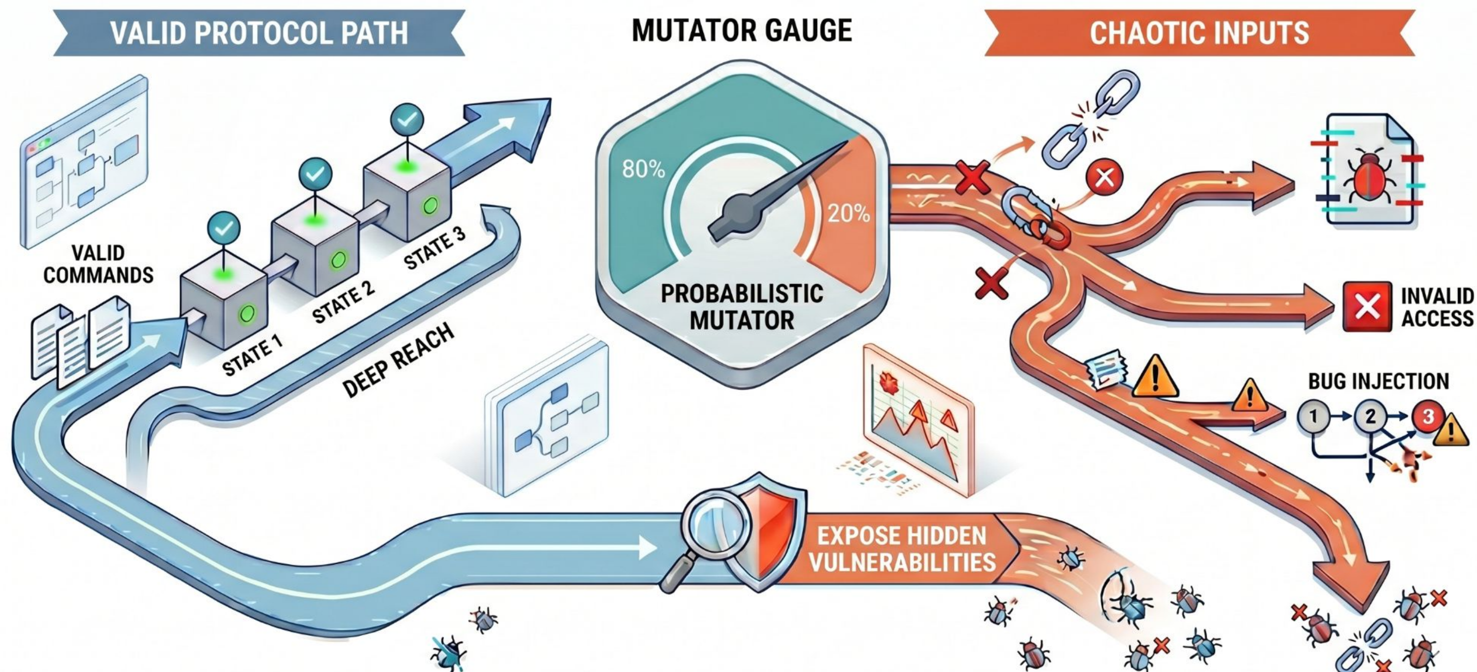
- The Fuzz Writer translates protocol knowledge into a new set of high-level commands. This is the concrete output of the AI's design process.

```
enum xhci_fuzz_cmds {
/* Host Controller Control */
OP_XHCI_HC_RESET = 10,

/* Asynchronous Command TRBs */
OP_XHCI_CMD_ENABLE_SLOT = 20,
OP_XHCI_CMD_ADDRESS_DEVICE,
OP_XHCI_CMD_CONFIGURE_ENDPOINT,

/* Direct Endpoint & Memory Control */
OP_XHCI_RING_EP_DOORBELL = 30,
... };
```

Balancing Structure and Chaos: the problem

- An AI pipeline that only generates valid, protocol-conforming code builds an excellent tool.
- **The Problem:** It only tests the "happy path".
- **The Reality:** Security vulnerabilities rarely hang out in well-behaved code. They hide deep within poorly tested error conditions, edge cases, and unexpected states.
- If we completely "tame" the fuzzer to stick strictly to the rules, we won't find the bugs that break the system.

Balancing Structure and Chaos: the solution



Impact & Future

What "Better Coverage" Means

- The new fuzzers don't just hit more lines; they behave more like a real OS
- Achieves **Full Device Initialization:** Successfully activates the entire QEMU USB stack, from controller setup to device enumeration
- **Exercises Core Logic:** High activity in critical functions like *xhci_process_commands* proves we are testing the main command pathways
- **Generates Valid Command Structures:** Constant activity confirms the fuzzer is correctly chaining Transfer Request Blocks (TRBs) that the device can execute

Results & Impact

- Automation **Velocity**: Generating stable, effective fuzz targets in hours, not weeks
- Path to >80% Coverage: This scalable approach puts full-device fuzzing coverage within reach
- Effective Vulnerability Discovery: The pipeline isn't just generating code; it's generating fuzzers that **find real, high-impact bugs**
- Democratizing Security: **Lowers the barrier** to entry for deep, protocol-aware fuzzing

Future Work & The Bigger Picture

- Continuing to Scale: Apply the pipeline to the remaining 200+ QEMU devices
- Smarter Loops: Enhance the feedback loops to understand runtime errors and crash triage
- Beyond QEMU: This agentic pipeline is a generic framework adaptable to other complex C/C++ projects like the Linux Kernel, browsers, or network daemons

Key Takeaways

- Fuzzing stateful protocols requires domain knowledge, which is a scaling bottleneck.
- An agentic AI pipeline (Expert + Writer) can automate the creation of this domain-specific knowledge.
- Self-correcting feedback loops are critical for turning raw AI output into robust, working tools.

Thank You

Questions?

navidem@google.com