
MicroVMM: AI-Assisted Verification-Oriented Virtual Machine Monitor¹

Thanh Do

Independent Researcher

With

Apart Research

Abstract

Modern agentic systems increasingly execute untrusted AI-generated code inside lightweight sandboxing environments. While many existing sandboxes rely on operating-system namespaces, recent advances in autonomous vulnerability discovery have exposed weaknesses in these traditional isolation boundaries. At the same time, vulnerabilities in virtualization infrastructure itself demonstrate that lightweight VMMs remain difficult to reason about formally due to complex shared-memory device protocols such as VirtIO. This project explores whether AI-assisted workflows can help construct formally analyzable virtualization software centered around explicit protocol invariants. Motivated by CVE-2026-5747, we derive queue lifecycle invariants from the vulnerability and construct a minimal verification-oriented VMM in Lean 4. We define abstract operational semantics for queue lifecycle behavior and establish refinement proofs connecting executable implementation behavior to the abstract model. The resulting system proves several safety properties, including post-activation queue immutability, directly eliminating the motivating vulnerability class within the modeled semantics. Our results suggest that AI-assisted development workflows can reduce the engineering complexity of refinement-oriented systems verification when combined with intentionally verification-oriented architectural design.

¹ Research conducted at the [The Secure Program Synthesis Hackathon](#), May 2026

1. Introduction

Virtualization has become a foundational component of modern computing infrastructure. Hypervisors and lightweight virtual machine monitors are widely used across cloud platforms, serverless systems, browser isolation, CI/CD pipelines, and multi-tenant execution environments. *Virtual machine monitors* (VMM) such as *Firecracker*, *QEMU*, and *crosvm* increasingly serve as security boundaries for executing untrusted workloads.

Historically, sandboxing systems relied heavily on operating-system isolation primitives such as user namespaces, seccomp, and capability restriction. More recently, many agentic execution environments have shifted toward lighter-weight runtime sandboxes based on JavaScript isolates or WebAssembly runtimes due to lower overhead and improved scalability. These approaches are now common in AI-assisted coding systems and dynamically generated execution pipelines.

However, the emergence of highly capable autonomous security-analysis models has significantly changed the threat model surrounding such sandbox boundaries. In early 2026, Anthropic disclosed Claude Mythos Preview, a frontier model reportedly capable of autonomously discovering and exploiting large numbers of previously unknown vulnerabilities across operating systems and browsers (Carlini et al., 2026). Due to these capabilities, the system was withheld from public release and deployed only through a restricted defensive-security initiative.

At the same time, multiple vulnerabilities were disclosed in AI-agent sandboxing systems themselves, including sandbox escapes involving configuration injection and symlink-based boundary violations (Zero Day Initiative, 2026); not to mention the rise of AI-assisted kernel privilege escalation attacks. These developments suggest that traditional runtime and usersn-based sandboxing may become increasingly fragile against highly capable automated adversaries.

As a result, lightweight virtualization systems with smaller trusted computing bases have become increasingly attractive for agentic sandboxing. Yet virtualization software itself remains difficult to reason about formally. VirtIO protocols are particularly security-sensitive because they expose guest-controlled descriptor queues directly to host-side virtualization code through shared-memory interfaces whose correctness depends on subtle lifecycle invariants.

This work is motivated by CVE-2026-5747 (AWS Security, 2026), a vulnerability affecting Firecracker’s VirtIO PCI transport implementation that was reportedly identified through Mythos-assisted analysis workflows (Carlini et al., 2026). The vulnerability involved post-activation mutation of queue configuration state, violating assumptions established during queue validation and potentially enabling unsafe memory behavior. Although not demonstrated as a full virtual machine (VM) escape, the bug exposed precisely the kind of implicit lifecycle assumption that becomes increasingly dangerous in high-throughput agentic execution systems.

We explore whether AI-assisted workflows can help construct formally analyzable virtualization software centered around explicit machine-checkable protocol invariants. Rather than retrofitting verification onto an existing hypervisor, we construct a minimal verification-oriented VMM in Lean 4 implementing a simplified VirtIO PCI transport and virtio-rng device together with refinement-based proofs over queue lifecycle semantics.

Our main contributions are:

1. A minimal, executable *virtual machine monitor* (VMM) written in Lean 4 that is capable of booting Linux, with a working virtio-rng device implemented over MMIO and PCI transports. This is the first public VMM written in Lean that we are aware of.
2. An abstract operational semantics for queue lifecycle behavior derived from a real-world VirtIO vulnerability.
3. Refinement-based proofs connecting executable implementation behavior to abstract queue semantics.

2. Related Work

Formal verification of systems software has historically focused on kernels and trusted execution substrates. The seL4 project (Klein et al., 2010) demonstrated machine-checked refinement verification of a production-capable microkernel from abstract specification to executable C implementation and established the importance of proof-oriented architectural simplification. The seL4 verification effort repeatedly emphasized that implementation structure and proof architecture must co-evolve in order to control verification complexity. Our work adopts a similar philosophy but targets virtualization protocol semantics rather than operating system correctness.

While seL4 established strong guarantees for the kernel itself, verification-oriented virtualization stacks and user-level virtual machine monitors remain comparatively underexplored. Prior work within the seL4 ecosystem (de Matos et al., 2022) has noted that VMM support remains fragmented and significantly less mature than the verified kernel substrate itself. Existing production virtualization systems such as QEMU and Firecracker prioritize performance and compatibility, resulting in highly stateful implementations that are not trivial to reason about formally. In contrast, our work intentionally constructs a minimal reference VMM designed around refinement-oriented verification and explicit queue lifecycle semantics.

Existing VirtIO assurance work has largely emphasized testing, fuzzing, symbolic execution, and bounded model checking (Stepanov, 2019). Recent Rust ecosystem efforts have explored proving localized queue-handling properties using tools such as Kani (Kumar, 2023). Our work instead focuses on refinement-style verification connecting executable implementation behavior directly to abstract queue semantics within Lean4.

The motivating vulnerability, CVE-2026-5747, exposed unsafe post-activation queue configuration mutation in Firecracker’s VirtIO PCI transport implementation. We derive our central queue activation invariants directly from this vulnerability class.

More broadly, our work intersects with emerging research on AI-assisted systems engineering and automated vulnerability analysis. Rather than treating large language models as autonomous theorem provers, we instead use them as assistants for implementation scaffolding, invariant extraction, theorem decomposition, and proof-oriented architecture design. Our broader goal is to evaluate whether AI-assisted workflows can reduce the engineering complexity associated with constructing refinement-oriented virtualization software.

3. Methods

We construct a minimal verification-oriented VMM in Lean 4 (hereafter *Lean*) implementing a simplified VirtIO PCI transport together with split virtqueue support and a `virtio-rng` device. The RNG device was intentionally chosen for its simplicity, which would fit well to the duration of this hackathon; however, it is still interesting enough to exercise various aspects of the VirtIO transport. The VMM architecture is intentionally constrained for formal tractability through single-threaded execution, explicit queue state transitions, and reduced mutable state. Rather than separating specification from implementation, the executable VMM itself is written directly in Lean, allowing operational semantics and proofs to coexist within the same environment.

For the formal verification component of this project, we derive queue activation invariants directly from the Firecracker VirtIO PCI vulnerability. In particular, we formalize the following invariant:

Once a queue has been activated and its configuration latched, subsequent guest writes must not modify descriptor-related fields of the active queue state.

We additionally prove a stronger refinement property of *immutability*: post-activation guest writes are restricted to shadow bookkeeping state and cannot modify either the live queue state used by the device or the latched configuration captured during activation. This establishes an explicit immutability boundary between validated queue configuration and subsequent guest-controlled updates within the modeled semantics.

The current proof architecture follows a layered refinement style in which low-level queue well-formedness properties are established first and then propagated into higher-level queue manipulation routines. In particular, the implementation emphasizes explicit modeling of queue activation boundaries motivated by the Firecracker vulnerability.

Development was performed using an AI-assisted workflow in which large language models were used interactively for implementation scaffolding, invariant exploration, theorem decomposition, and proof engineering assistance. However, correctness derives entirely from machine-checked proofs within Lean rather than language model outputs themselves.

Trusted Computing Base

Our refinement proofs establish correctness properties over the Lean implementation and its abstract VirtIO queue semantics. However, several components remain within the trusted computing base.

Most notably, the executable VMM communicates with Linux KVM through a *thin C shim layer* that is not currently verified. This shim layer is kept intentionally minimal: only KVM IO controls, C struct manipulation, and output handling were implemented there (we accidentally found a Lean runtime crash while working on this project). Correctness therefore assumes faithful transmission of VM state, queue configuration, memory mappings, and VM-exit behavior across this boundary.

The current work does not attempt formal verification of Linux KVM itself. Recent efforts such as SeKVM (Li et al., 2021) demonstrate that verification of KVM virtualization layers can be achieved, so this opens up an interesting direction for future work.

Our guarantees additionally rely on correctness of the Lean compiler and runtime system, the host operating system kernel, and underlying hardware execution semantics. We do not currently model verified compilation, verified foreign-function interfaces, or hardware-level memory consistency behavior.

Consequently, the refinement proofs should be interpreted as establishing correctness of the modeled VirtIO queue semantics relative to the trusted execution environment rather than end-to-end hypervisor correctness.

4. Results

We successfully brought up a Linux 7.0 guest on our Lean-based microVMM and established interactive communication through the serial console. The implemented virtio-rng device was verified to function correctly with an unmodified Linux guest, demonstrating interoperability between the simplified Lean VirtIO implementation and a production operating system.

Reproducibility instructions are provided in the repository README, where the guest can be launched on a Linux x86_64 host and tested using the provided `virtio-rng-read` script.

On the verification side, we mechanically formalized the queue activation invariants derived from the Firecracker vulnerability class described earlier in the report (this is [row #4 in the guide](#)). Using an AI-assisted workflow based on GPT-5.4 through GitHub Copilot, we refined substantial portions of the proof infrastructure down to the trusted shim boundary consisting of the thin C FFI layer used for KVM interaction. The current proof effort covers both PCI and MMIO transport paths within the modeled semantics. All specifications, invariants, and proof artifacts remain independently checked by Lean and reviewed by a human.

Beyond the primary activation-safety theorem, we additionally mechanized several supporting queue correctness properties including descriptor bounds preservation, queue structural well-formedness, and activation-state consistency. Due to space limitations, a more detailed description of the current proof state and theorem inventory is provided separately in the [Proof Engineer Guide included with the repository](#).

5. Discussion and Limitations

The emergence of systems such as Claude Mythos suggests a future in which vulnerability discovery, exploit generation, and protocol analysis become increasingly automated. At the same time, recent vulnerabilities in agentic coding platforms and sandboxing infrastructure demonstrate that runtime-based isolation boundaries are often fragile under adversarial pressure. Together, these trends indicate that AI systems may increasingly become both the consumers and discoverers of vulnerabilities within the infrastructure used to contain them.

In this environment, stronger isolation boundaries with smaller trusted computing bases become increasingly important. Lightweight virtualization platforms provide one promising direction because they rely on hardware-assisted isolation rather than purely runtime-level separation. However, our work also highlights that virtualization software itself remains vulnerable to subtle protocol lifecycle errors that are difficult to identify through testing alone.

A key implication of this work is that future AI safety infrastructure may require substantially more formal reasoning about low-level systems behavior than is currently common in mainstream sandbox engineering. As AI systems become more capable at exploring state spaces and exploiting unintended behaviors, informal assumptions embedded within systems software become increasingly dangerous.

Our results additionally suggest an important asymmetry: AI systems may accelerate both offensive vulnerability discovery and defensive verification workflows simultaneously. In our experience, large language models were effective not as autonomous theorem provers, but as assistants for implementation scaffolding, invariant extraction, and proof-oriented architecture exploration. This indicates that AI may lower the cost of constructing formally analyzable systems even if machine-checked proofs remain essential for trust.

More broadly, we observe a convergence between AI safety and systems verification. Historically, formal methods were often viewed as too expensive or specialized for practical deployment in systems infrastructure. However, if autonomous AI systems substantially increase the rate and sophistication of vulnerability discovery, the economic tradeoff may shift toward architectures intentionally designed around machine-checkable invariants and refinement-oriented reasoning.

Limitations

Our implementation intentionally models only a restricted subset of VirtIO semantics and therefore does not constitute comprehensive virtualization verification. In particular, the current model excludes packed virtqueues, MSI-X interrupts, weak memory ordering, multiqueue semantics, migration behavior, DMA consistency modeling, and concurrent queue execution. Also, some interesting safety properties like guaranteed completion are not modelled or modelled only partially. As a result, the proven properties apply only within the simplified operational semantics captured by the current implementation.

The correctness of the refinement proofs additionally depends on the adequacy of the abstract queue semantics themselves. Like all formal verification efforts, our proofs establish correctness relative to the modeled semantics rather than correctness relative to all possible real-world hardware behavior. Unmodeled behaviors or omitted protocol interactions may therefore invalidate assumptions underlying the refinement relation.

Although refinement proofs establish a strong correspondence between executable implementation behavior and abstract queue semantics, correctness remains relative to the trusted execution base including the Lean runtime, extraction environment, underlying C compiler, host operating system, and hardware execution semantics. Our work does not attempt verified compilation, verified hardware modeling, or end-to-end machine-code correctness.

The verification-oriented architecture itself also introduces external validity concerns. Our implementation intentionally prioritizes proof tractability through reduced concurrency, simplified transport semantics, and explicit lifecycle state machines. Production virtualization systems frequently adopt substantially more complex architectures optimized for performance, compatibility, and scalability. Consequently, the techniques explored here may not transfer directly to production-scale hypervisors without substantial additional proof engineering effort.

The AI-assisted development workflow introduces additional methodological risks. Large language models may hallucinate protocol semantics, omit important invariants, or bias abstraction selection toward incomplete formalizations. Although machine-checked proofs ultimately determine correctness, human developers remained necessary throughout the development process for semantic validation, abstraction refinement, and proof interpretation.

Finally, the vulnerability motivating this work represents only one category of virtualization failure. Other classes of vulnerabilities involving speculative execution, timing behavior, DMA aliasing, or concurrent state interaction may require substantially different modeling and verification techniques.

Future Work

Several important directions remain for future work. One immediate extension is expansion of the operational semantics to support additional VirtIO functionality including packed virtqueues, interrupt delivery, concurrent queue execution, and DMA consistency modeling. Extending the refinement framework to weak memory semantics and multiqueue behavior would substantially improve realism relative to production virtualization systems.

Another major direction is reducing the remaining semantic gap between the Lean reference implementation and production VMM codebases. In particular, a promising path is integration with Aeneas (Ho et al., 2022), a verification framework for Rust programs that translates Rust code into formal representations suitable for theorem provers. Aeneas has been developed specifically to support verification-oriented reasoning over real Rust implementations while preserving correspondence with executable source code. Because systems such as *Firecracker* and *crosvm* are implemented largely in Rust, Aeneas may provide a practical mechanism for connecting production VirtIO implementations with the abstract queue semantics developed in this work. Rather than rewriting production hypervisors directly in Lean, future work could establish refinement relations between verified abstract queue models and extracted representations of real Rust VMM components. This would substantially strengthen external validity while preserving many advantages of refinement-oriented verification.

Additional future work includes verification of more complex VirtIO devices such as block and network interfaces, as well as automated extraction of protocol invariants from vulnerability reports, patches, and execution traces. More broadly, we believe AI-assisted workflows may be particularly effective at accelerating exploration of proof-oriented systems architectures designed around machine-checkable semantics rather than replacing formal verification itself.

6. Conclusion

We presented a refinement-oriented verification approach for a minimal VirtIO sandbox VMM implemented in Lean. Motivated by a real-world Firecracker VirtIO PCI vulnerability identified through AI-assisted analysis, we derived explicit queue lifecycle invariants and constructed executable operational semantics together with refinement proofs connecting implementation behavior to abstract protocol models.

Our implementation demonstrates that virtualization software can be designed directly around machine-checked semantics while preserving meaningful implementation-level correctness guarantees. At the same time, our experience suggests that AI-assisted workflows may substantially reduce the engineering friction associated with proof-oriented systems development when combined with intentionally verification-friendly architectural design. Ultimately, this work argues that formal methods become significantly more practical when systems software is

constructed around explicit invariants from the outset rather than verified only after implementation complexity has accumulated.

Code and Data

- **Code repository:** <https://github.com/chitoge/lean4-microvmm>
- **Prebuilt artifacts:** <https://github.com/chitoge/lean4-microvmm/releases/tag/v0.0.1>
- **Accompanying slides:** <https://nyanwith.me/projects/microvmm/slides.pdf>

References

1. Klein, G., et al. (2010). “seL4: Formal Verification of an Operating-System Kernel.” *Communications of the ACM*.
<https://cacm.acm.org/research/sel4-formal-verification-of-an-operating-system-kernel/>
2. de Matos, E., et al. (2022). “seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM.” *Electronics*, 11(24), 4201.
<https://www.mdpi.com/2079-9292/11/24/4201>
3. Carlini, N., Cheng, N., Lucas, K., Moore, M., Nasr, M., Prabhushankar, V., Xiao Angulu, W., Ben Asher, E., Bow, J., Bradwell, K., Buchanan, B., Gaynor, A., et al. (2026). “Assessing Claude Mythos Preview’s Cybersecurity Capabilities.” Anthropic Red Team / Project Glasswing. <https://red.anthropic.com/2026/mythos-preview/>
4. AWS Security. (2026). “CVE-2026-5747 - Out-of-bounds Write in Firecracker virtio-pci Transport.” AWS Security Bulletin 2026-015-AWS.
<https://aws.amazon.com/security/security-bulletins/rss/2026-015-aws/>
5. Ho, S., & Protzenko, J. (2022). “Aeneas: Rust Verification by Functional Translation.” *arXiv preprint arXiv:2206.07185*. <https://arxiv.org/abs/2206.07185>
6. Li, S. et al. (2021). “Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor.” *30th USENIX Security Symposium (USENIX Security 21)*.
<https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei>
7. Stepanov, D. (2019). “Virtio Device Fuzzing.” *KVM Forum 2019*.
<https://www.youtube.com/watch?v=dk6SUD8ovXw>
8. Kumar, R. (2023). “How Open Source Projects are Using Kani to Write Better Software in Rust.” AWS Open Source Blog.
<https://aws.amazon.com/blogs/opensource/how-open-source-projects-are-using-kani-to-write-better-software-in-rust/>

LLM Usage Statement

We used GPT-5.4 through GitHub Copilot as a work orchestrator, implementation and proof partner in this project. Additionally, we used ChatGPT as a reviewer and editor for the report. All specs and proofs were checked, both manually by the author and partially as part of automated CI on GitHub. Ideas, references, architecture, design decisions, and claims are the author’s own.