# Tractable Problems in AI Security via Formal Methods

**Quinn Dougherty**
Forall R&D
quinn@for-all.dev

**Max von Hippel**
In an independent capacity
maxvh@hey.com

**Nora Ammann**
ARIA
nora.ammann@aria.org.uk

**Gregory Malecha**
Skylabs AI
gregory@skylabs-ai.com

**Abstract.** Secure program synthesis is popping off in 2026 [1], [2], [3], which will be great for our overall cyber resilience. However, its not obvious that it will actually be applied to AI security in real life. To seize this opportunity, we need to map out the relevant layers in the current ML inference and training stack and figure out what widgets represent formal methods opportunities. We are not alone in this read: a June 2026 RAND expert survey [4] lands on the same split we do — formal methods can secure the infrastructure (memory safety, access control, sandbox isolation) even though they cannot adjudicate model semantics like jailbreaks — and ranks cryptographic primitives and access control as the highest-leverage place to start. This document turns that diagnosis into shovel-ready specs. The June 2026 executive order on AI innovation and security [5] points the same way — voluntary vulnerability scanning, hardened critical infrastructure, federal funding for AI vulnerability detection — while declining to mandate licensing or preclearance; what it does not say is how any of that earns its assurances. The demand side is on record too: Anthropic's frontier safety roadmap commits to "leveling up across the board" by mid-2027 — hardened `Kubernetes` access controls, allowlist-based egress, supply-chain integrity for builds, short-lived cryptographic identities [6] — a checklist that reads layer-for-layer like the stack this document maps. Let us treat ML training and inference infrastructure with the seriousness we treat airplanes; and on the software side, that will mean at least some formal methods.

## Contents

# 1 Introduction

## 1.1 How to read this document

The document has two halves. Section 2 walks the five layers of the ML training and inference stack — execution harness, software/ML framework, orchestration and cloud, firmware and low-level systems, hardware and physical supply chain — and, for each, sketches the status quo and how attackable it is. Section 3 is the payload: a list of concrete problems, each tagged by which layers of the stack it touches and whether it is an *enabler* (unbottlenecks a class of downstream work) or a *widget* (a scoped, shippable artifact). The website at tractable.for-all.dev mirrors the same content, with the layer/problem tagging exposed as a many-to-many filter.

## 1.2 What formal methods can and can't claim

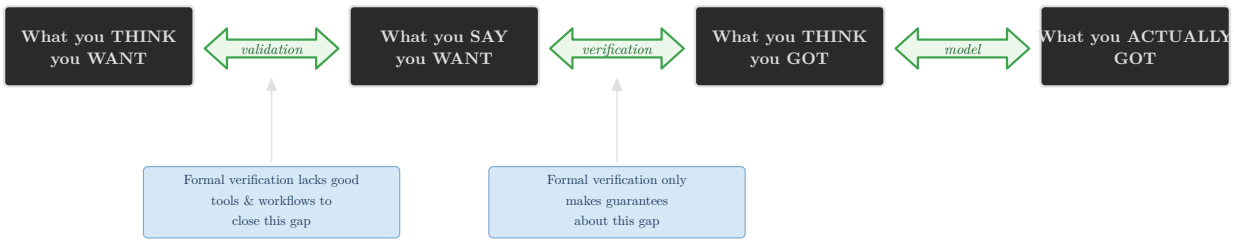


Figure 1: After Evan Miyazono. Formal verification only closes the middle gap; the elicitation gap on the left is out of scope for the proof itself, and the modeling gap on the right is only as good as the model of the system the proof is stated against. Both ends are where most real-world failures live.

This is the same line the June 2026 RAND expert survey draws: verification reaches code-level correctness — memory safety, access control, protocol enforcement, sandbox isolation — and has little purchase on model semantics like jailbreaks or sandbagging [4]. We take that boundary as our scope. Everything in this document sits to the left of the modeling gap, in the infrastructure the model runs on, not in the model itself.

## 1.3 What we count as tractable

Three working principles narrow the problem list in Section 3.

### 1.3.1 Mechanism, not policy

Verification has the most leverage on the mechanisms that establish system integrity — microkernels, hypervisors, drivers, network protocols — and the least on the policies layered on top of them. Container scheduling on `Kubernetes` is the canonical example: a scheduler that places pods optimally produces a faster answer, not a more correct one, and verifying its placement decisions does not improve isolation between tenants if the underlying runtime is unsound. We accordingly scope the orchestration layer (Section 2.3) to its mechanism content — distributed-protocol correctness, IAM logic, network-fabric isolation, runtime confinement — and treat scheduler optimization as out of scope. The same line shows up at every layer: there is a verifiable mechanism underneath, and a policy or optimization above it whose correctness is downstream of the mechanism's.

### 1.3.2 Small API surfaces are reachable; large ones are not

Compare NOVA's 16-hypercall interface [7] to the POSIX surface that Linux exports, or to the API a Docker daemon exposes through its versions. The first is small enough to specify and prove against; the others balloon with each release and have no single coherent specification to target. Tractability tracks API size more reliably than it tracks codebase size — a small interface in front of a large implementation is a verification target, a large interface is not. This shapes the widget specs in Section 3 toward the smallest customer-relevant subset — the 10 or 25 functions a real workload actually uses — rather than full-coverage proofs of sprawling APIs. The RAND survey points at the same first targets from the other direction: its experts ranked cryptographic primitives and access control as simultaneously highest in security value and highest in verification feasibility, and noted that production-ready verified implementations of both already exist [4].

### 1.3.3 Separable, language-agnostic specifications

A system's specification should be separable from its implementation, both technically (so a Rust rewrite of a C++ component does not invalidate the proof effort) and legally (so an open spec can be referenced by implementations under stricter licenses). NOVA is the existence proof: the implementation is GPL-2 (Intel and TU Dresden) while the specification is licensed separately under Blue Rock [7]. The same separation is what would let a Linux retrofit happen without forcing every maintainer onto a single proof toolchain. We treat separability as a precondition: a widget that cannot factor cleanly into spec-and-implementation is one we cannot recommend, regardless of how shippable the implementation looks. A related cut shows up in the RAND recommendations along a different axis — split the verified invariant-enforcing core from the unverified performance-optimized code around it, so verification can keep pace with a fast-moving codebase instead of holding the whole thing hostage to its slowest-changing part [4].

## 1.4 Why this is hard to fund

The honest difficulty: formal methods compete in a market where the unverified alternative is usually free. A startup considering whether to buy a verified hypervisor over KVM, or a verified container runtime over runc, is comparing a paid product against a zero-marginal-cost open-source incumbent that is good enough for the threat model the customer thinks they have. The ROI calculation is upside-down before the conversation starts. This is the central reason the formal-methods talent that exists today is concentrated in domains — avionics, defense, automotive [8] — where regulators force the comparison to be against a counterfactual incident rather than against the free alternative.

The document does not solve this problem, but it tries to make the right cases visible. For each tractable problem in Section 3, we name the threat model the verified component would close, the alternatives a buyer is implicitly comparing it against, and the lift to deliver a usable artifact rather than a research prototype. AI infrastructure is one of the few private-sector settings where the counterfactual incident is large enough to invert the calculation — model weight exfiltration, training-data poisoning, or container escape from an agentic workload are losses on the order of the model's training cost — and where the customers (frontier labs, regulated downstream deployers) have both the budget and the risk model to act on it. Those losses split along the two converging threat models a 2026 RAND expert consultation puts at the center of the case for verifying this infrastructure — a conventional cyber adversary (nation-state or criminal) stealing

weights or disrupting operations, and the loss-of-control case where a misaligned model exploits vulnerabilities in its *own* infrastructure to bypass safety monitors or exfiltrate itself [4] — and one hardened runtime closes both. That same consultation is a useful corrective on how the sale actually closes: in a capability race, frontier labs will not eat a meaningful performance or velocity penalty for security alone, so a verified component has to win on something they already want — fewer bugs, breach protection, faster incident recovery — with the assurance riding along rather than carrying the pitch. The same report offers the existence proof that this is achievable: AWS's Automated Reasoning Group found that formally verifying parts of S3 left the code often *more* performant than what it replaced, easier to maintain, and faster to release [4] — verification buying velocity rather than spending it, which is what inverts the upside-down ROI above. Public money is now in play too: the June 2026 executive order directs OMB to identify federal grant funding for AI vulnerability detection [5], a channel that could underwrite the early widgets in Section 3 before any private buyer's ROI flips. The labs' own security planning corroborates the diagnosis: Anthropic's frontier safety roadmap, under the heading "leveling up across the board," names the same surfaces this document does — hardened `Kubernetes` access controls across frontier clusters, allowlist-only network egress in sensitive environments, integrity guarantees for build sources and dependencies, cryptographically verified short-lived identities [6]. The roadmap states those as configuration goals a security team checks; they are also properties a verified component could discharge rather than attest. Making the business case requires actually writing it down. That is what this document is.

## 2 The ML Inference and Training Stack

The five layers below run top to bottom in the order an adversary walks the stack. Each section that follows opens on the layer's status quo and current attack surface, then surfaces the FM-shaped widgets that would close part of it. Scope follows Section 1.3 — infrastructure only; the model itself is out of frame.

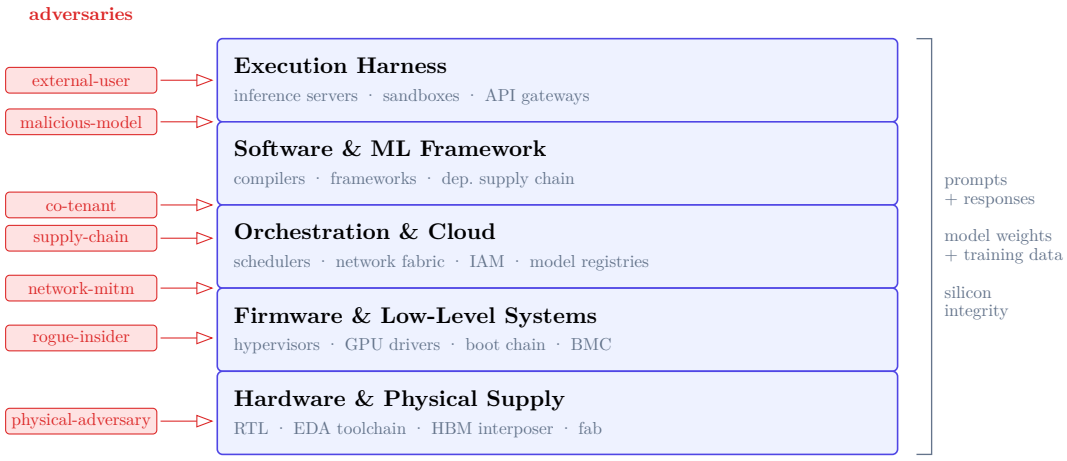


Figure 2: The five layers of the ML inference and training stack, with the adversary archetypes (left) that each layer invites and the assets (right) that pass through them. Adversaries are tagged on each layer below using the same labels; see Section 4 for full descriptions.

## 2.1 Execution Harness

*Related problems:* Adversarial Robustness of Formal Methods · Specification Elicitation and Validation · OCI Runtime Hardening · AI Control via Proof-Carrying Code · Edge Policy Verification · Weight Integrity and Kernel Supply Chain · Sampler Integrity and Determinism · Verified Input Parsers · Verified Protocol Trust Boundaries · Context Window Integrity Under Adversarial Length · Capability-Safe Tool Interfaces · Audit Log Integrity and Session State Channels · Governance and Interpretability of Neuralese Proof Stacks · Formal Theory of Weird Machines in Agents

*Invites:* External User · Malicious Model · Co-tenant

The execution harness is the outermost software layer of an ML deployment: the wrapper that actually calls the model and handles its inputs and outputs. A user's prompt enters here, gets tokenized and batched, passes through the model, and returns as a completion. Because every interaction transits this layer, it is simultaneously the easiest place to bolt on security controls and the place where a failure is most directly exploitable from the outside.

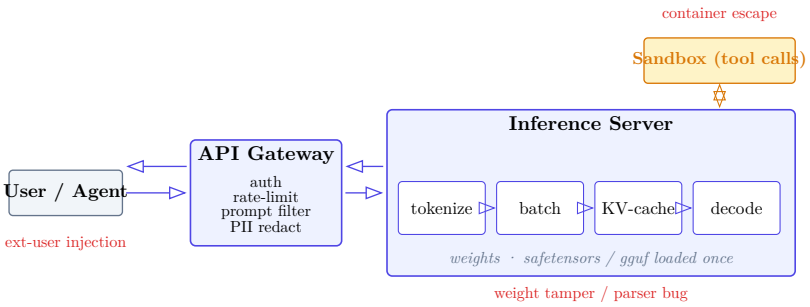


Figure 3: A typical request flow through the execution harness. The gateway makes policy decisions on the way in and the way back; the inference server holds the hot path; tool calls divert into a sandbox. The labelled red callouts mark the externally-reachable failure modes addressed later in Section 3.

Three components make up a typical execution harness, each with its own attack surface.

### 2.1.1 Inference Servers

An inference server (`vLLM`, Hugging Face `TGI`, Nvidia `Triton`) manages GPU memory, batches incoming requests for throughput, and exposes the model behind an HTTP or gRPC endpoint. The server is responsible for continuous batching, KV-cache management, and (increasingly) speculative decoding — all performance-sensitive paths written in a mix of Python and C++/`CUDA`.

The security-relevant surface here is large. The server deserializes model weights, often from formats like `SafeTensors` or `GGUF`, and the parsers for the latter have had their own history of memory-corruption bugs [9] (`SafeTensors`' own audit [10] found no critical flaws, but the `GGUF` loader in `llama.cpp` has seen repeated integer-overflow-to-heap-overflow patterns). It accepts user-controlled input (prompts, generation parameters, `LoRA` adapter names) that flows deep into the batching and memory-allocation logic. A malformed request that triggers an out-of-bounds KV-cache write, for instance, could corrupt another user's in-flight completion in a multi-tenant deployment. Formal methods work in this area has two legs: memory-safety of the batching and caching layers, and verified parsers across the several deserialization boundaries the server straddles (weights on the way in, request payloads on the hot path, and structured tool-call outputs on the way back). The second leg is the subject of Section 3.13; the tooling is mature and the gap is targeting, not capability.

Inference proxies like `LiteLLM` are also part of the dependency supply chain (Section 2.2.3) — a compromised package at this layer gives an attacker access to every API key and model endpoint the proxy routes.

### 2.1.2 Sandboxes

When an AI system can execute code — a "code interpreter" in a chatbot, a tool-use agent writing and running scripts — it needs a sandbox. In practice this means a containerized environment (often an OCI container or a microVM like `Firecracker`) with a constrained syscall profile and filesystem. The sandbox must prevent the AI-generated code from escaping to the host, exfiltrating data from other tenants, or persisting state across invocations in unintended ways.

The gap between "container" and "sandbox" is where real attacks live. Default container configurations are not security boundaries; they share a kernel with the host. A sandbox that allows `ptrace`, or that mounts the `Docker` socket, or that runs with `CAP_SYS_ADMIN`, is a sandbox in name only. The OCI runtime hardening problem (Section 3.5) addresses this directly: writing machine-checked specifications for what a sandbox policy must guarantee and verifying that a given runtime configuration meets them.

### 2.1.3 API Gateways

The API gateway sits at the public edge. It handles authentication, rate limiting, content filtering, prompt injection detection, and usage metering. In many deployments this is a conventional reverse proxy (`Envoy`, `Kong`) with AI-specific middleware bolted on: a classifier that scans prompts for injection attempts, a filter that redacts PII from completions, a logger that records interactions for abuse review.

The security challenge is that these filters are making semantic decisions about adversarial input using heuristics or secondary ML classifiers — a brittle arrangement. A prompt injection that evades the filter is not a bug in the filter's code but a failure of its classification boundary. Formal methods have limited traction on the classification problem itself, but they can verify the *plumbing*: that the gateway's policy engine correctly composes its rules, that a request flagged by the filter cannot reach the model through an alternate code path, and that rate-limiting state cannot be poisoned by concurrent requests. The value here is ensuring that when a policy says "block this," the infrastructure actually does.

Beneath all of this AI-specific middleware sits the protocol layer that admits the request in the first place: the TLS session terminating the HTTPS connection, the SSH daemon an operator reaches the box through, and increasingly the WireGuard overlay that decides which machines can see the fleet at all. These are the oldest and most-studied trust boundaries in the stack, and the ones whose single-bug blast radius is largest; Section 3.14 treats them as a joint verification target.

## 2.2 Software and ML Framework Layer

*Related problems:* Adversarial Robustness of Formal Methods · Specification Elicitation and Validation · AI Control via Proof-Carrying Code · Weight Integrity and Kernel Supply Chain · Sampler Integrity and Determinism · Verified Input Parsers · Governance and Interpretability of Neuralese Proof Stacks

*Invites:* Supply Chain · Malicious Model · Rogue Insider

Between the orchestration infrastructure below and the execution harness above sits the software that actually defines and compiles a model: the frameworks researchers write against, the compilers

that lower their code to GPU kernels, and the sprawling dependency graph that connects everything. This layer is where the mathematical definition of a neural network meets the reality of executable code. A compromise here is dangerous because it can alter what a model *computes* without changing what it appears to be — the weights, the architecture description, and the training logs all look clean while the compiled artifact does something else.

### 2.2.1 Compilers

An ML compiler takes a high-level model description and lowers it through several intermediate representations to GPU machine code. The typical path is something like `PyTorch` → `torch.compile` → `Triton` IR → `LLVM` IR → `PTX` → `SASS`, though the specifics vary: `TVM` uses its own `Relay/TIR` stack, `XLA` compiles through `HLO` and `StableHLO`, and `JAX` traces through `jaxpr` before hitting `XLA`. Each stage applies optimization passes — operator fusion, memory layout transforms, quantization, tiling — that rewrite the computation in ways the model author never sees directly.

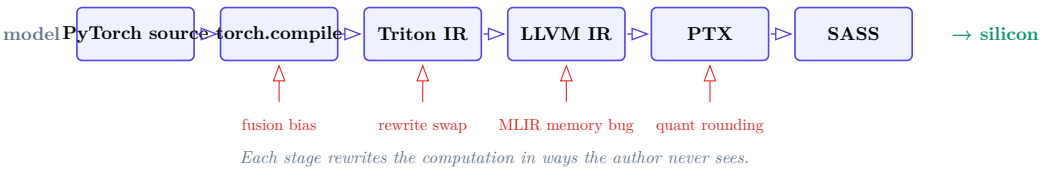


Figure 4: A typical `PyTorch` lowering path, with sample injection points (red) where a poisoned pass could alter what the silicon computes without changing what the source, weights, or training logs look like.

The attack surface is a modern instance of Thompson's "Trusting Trust" [11]. A poisoned compiler pass could inject a small additive bias into attention weights, swap an activation function under specific input conditions, or silently alter a quantization rounding mode. The result would be a model that behaves correctly on benchmarks but misbehaves on targeted inputs, with no trace of the modification in source code, model weights, or training logs. The compilation chain is long enough that auditing every stage by hand is not feasible.

This is not hypothetical. `LLVM`'s `MLIR` infrastructure — shared by `TVM`, `XLA`, and `Triton` — had multiple memory-management vulnerabilities in 2023 [12] where a crafted `MLIR` file could trigger crashes or worse. More telling is a 2025 `XLA`:TPU miscompilation in the approximate top-k operation that returned silently wrong results [13] for certain batch sizes, affecting production models. The compiler produced valid-looking output; it was just computing the wrong function. No error was raised. This is the failure mode that formal methods are built to prevent.

The formal methods community has begun to engage. Bhat et al. [14] formalized a core calculus for SSA-based IRs in `Lean` and verified peephole rewrites across three use cases: `LLVM` bitvector rewrites, structured control flow, and fully homomorphic encryption. Fehr et al. [15] defined semantics for five `MLIR` dialects, built three verification tools, and found five miscompilation bugs in upstream `MLIR` in the process. Both cover individual rewrites and local properties, not end-to-end compilation correctness. The gap between "verified peephole rewrites" and "verified compiler" is the same gap `CompCert` [16] closed for C — and closing it for even one ML compilation path (say, `StableHLO` to `PTX` for a fixed set of operations) would be a major result.

A practical intermediate target: verified compilation of attention kernels. Attention is the computational core of transformer inference, and it is also the operation most aggressively optimized by

ML compilers (`FlashAttention`, `PagedAttention`, various fused variants). Proving that the compiled kernel preserves the semantics of the source-level attention specification, up to documented floating-point tolerances, would cover the highest-value target in the compilation chain.

### 2.2.2 Frameworks

`PyTorch`, `JAX`, and `TensorFlow` are the environments where models are defined, trained, and exported. They provide automatic differentiation, GPU dispatch, a zoo of layer implementations, and serialization formats for saving and loading model weights. Each is a million-line codebase with native extensions, JIT compilers, and deep operating system integration, but the highest-impact vulnerability class is mundane: deserialization.

`PyTorch`'s default serialization uses Python's `pickle` protocol, which can embed arbitrary executable code in a saved file. Loading an untrusted `.pt` file is functionally equivalent to running `eval()` on attacker-controlled input. This was understood in principle for years, but [17] demonstrated that even the `weights_only=True` safeguard — `PyTorch`'s own recommended mitigation — could be bypassed, restoring full remote code execution. `Hugging Face` uses `PickleScan` to screen uploaded models, but `JFrog` found three zero-day bypasses in the scanner itself [18]: one corrupts magic-number detection so the scanner halts with an "invalid magic number" error while `torch.load()` still loads the payload; a second exploits a file-extension mismatch that skips scanning entirely; and a third uses submodule imports to downgrade the severity of dangerous globals from blocked to merely flagged. In a separate line of attack, `ReversingLabs`' nullifAI technique [19] hid working reverse shells in `PyTorch` models by 7z-compressing the `pickle` payload so the scanner's decompression failed, even as the malicious code ran first. `TensorFlow` has its own variant: `Keras` Lambda layers embed arbitrary Python in model definitions, and [20] showed that the `safe_mode` fix could be bypassed via a downgrade attack when loading older checkpoints.

The `SafeTensors` format, developed by `Hugging Face`, stores only tensor data and metadata with no code execution path. An independent security audit [10] found no critical flaws. This is the right direction, but the safety argument rests on an audit, not a proof. A serialization format with a machine-checked parser — in the style of `EverParse` [21] or similar verified parsers — would close the vulnerability class permanently rather than playing whack-a-mole with scanner bypasses.

Beyond serialization, the frameworks expose attack surface through JIT compilation and custom operators. `PyTorch`'s `torch.utils.cpp_extension.load()` compiles C++/`CUDA` code into `/tmp` and dynamically loads the resulting shared library. Any process that can influence the source or the cached `.so` gets native code execution. `TensorFlow` has accumulated roughly 435 CVEs across 44 CWE categories [22], many in parsing and memory management of its graph representation. Formal verification of core tensor operations — proving that `softmax`, `layernorm`, or `matmul` produce results consistent with their mathematical definitions across all compilation backends — remains open territory.

### 2.2.3 Dependency Supply Chain

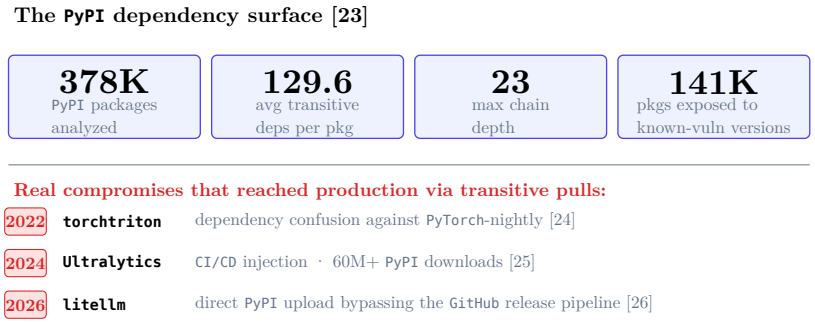


Figure 5: Top: ecosystem-level statistics on the transitive-dependency surface from [23]. Bottom: three real compromises that reached production ML stacks via transitive pulls in 2022–2026.

A typical ML project does not just depend on `PyTorch` or `TensorFlow`. It depends on a graph of packages — data loaders, tokenizers, experiment trackers, serving utilities, `CUDA` bindings — that pull in their own transitive dependencies. Mahon et al. [23] analyzed 378,573 `PyPI` packages and found that the average package has 129.6 transitive dependencies spanning a dependency chain up to 23 levels deep. A single CVE in `urllib3` created guaranteed exposure in 1,906 downstream packages. The ML ecosystem sits on top of all of this and adds its own layer of domain-specific packages with fast release cycles and small maintainer teams.

Supply chain attacks against this graph are not theoretical. In December 2022, an attacker uploaded a malicious `torchtriton` package to `PyPI` with the same name as `PyTorch`'s internal dependency [24]. Because `pip` resolves the public `PyPI` index before private indices by default, `PyTorch`-nightly users received the attacker's version, which exfiltrated hostnames, environment variables, and password files. In December 2024, the `Ultralytics` YOLO library (60+ million `PyPI` downloads) was compromised through a `GitHub Actions` script injection: the attacker crafted PR branch names that injected code into the CI/CD pipeline, and the published package bundled a cryptominer [25]. In 2026, the legitimate `litellm` package — an inference proxy pulled as a transitive dependency by many ML stacks — was compromised by a direct upload to `PyPI` that bypassed the project's `GitHub` release pipeline entirely [26]. The injected payload harvested SSH keys and cloud credentials and attempted lateral movement across `Kubernetes` clusters. The attacks are moving up the dependency graph, where one package reaches thousands of dependents.

The defenses that exist are largely unsigned and unverified. `SLSA` (Supply-chain Levels for Software Artifacts) [27] defines four levels of provenance assurance, from basic metadata (L1) through hermetic reproducible builds (L4). Most ML toolchains operate at L0 or L1 — packages are uploaded by individual maintainers from their laptops with no build provenance, no signed attestation, and no reproducibility guarantee. `Sigstore` [28] offers keyless artifact signing tied to OIDC identities, which would at least bind a package version to a specific CI run. But even with signed provenance, the resolver itself is a trust boundary: `pip`'s dependency resolution algorithm is complex, underspecified, and has produced dependency confusion vulnerabilities by design [29].

Formal methods can contribute at two levels. First, at the resolver level: proving that a dependency resolution algorithm correctly implements its specification and cannot be tricked by namespace collisions, version range manipulation, or index priority ordering. This is a bounded, well-defined verification target. Second, at the build level: verified reproducible builds where the mapping

from source to artifact is proven deterministic, so that any party can independently verify that a published package corresponds exactly to its claimed source. Achieving `SLSA` L4 with machine-checked guarantees — rather than relying on build system configuration being correct — would make supply chain substitution attacks structurally impossible rather than merely detectable.

## 2.3 Orchestration and Cloud Layer

*Related problems:* Specification Elicitation and Validation · OCI Runtime Hardening · AI Control via Proof-Carrying Code · Edge Policy Verification · Scheduler Integrity and Co-Tenancy Isolation · Fabric Policy Verification · Verified Network Tap · Verified Protocol Trust Boundaries · Audit Log Integrity and Session State Channels · Governance and Interpretability of Neuralese Proof Stacks · Formal Theory of Weird Machines in Agents

*Invites:* Co-tenant · Rogue Insider · Network MITM · Supply Chain

The orchestration layer manages the compute resources that ML workloads run on: scheduling jobs across GPU clusters, routing traffic between them, and controlling who can access what. It sits below the frameworks and above the firmware — a layer of distributed systems software that most ML engineers interact with only through configuration files, but whose security properties determine whether isolation between tenants, jobs, and data pipelines actually holds. Following Section 1.3.1 we treat this layer as a stack of mechanisms (runtime confinement, distributed-protocol correctness, network-fabric isolation, IAM logic) and not as a question about scheduler optimization; placement and load-balancing decisions are downstream of these mechanisms and out of scope.

### 2.3.1 Cluster Orchestration

GPU training and inference jobs run inside containers managed by `Kubernetes`, `Slurm`, or `Ray`. The container is the unit of isolation: it is supposed to confine a workload to its own filesystem, network namespace, and device access. In practice, GPU workloads erode this confinement. NVIDIA's container toolkit — the standard mechanism for exposing GPUs inside containers — had a critical TOCTOU vulnerability [30] that allowed a malicious container image to escape to the host, affecting 33% of cloud environments per Wiz's estimate. The underlying container runtime, `runc`, had its own escape [31] via leaked file descriptors. These are not exotic attacks; they are the kind of bugs that container runtimes accumulate routinely, and they interact badly with the privileged device access that GPU workloads require. The OCI runtime hardening problem (Section 3.5) addresses the gap between what orchestrators *assume* containers guarantee and what the runtimes actually enforce.

`Slurm`, the dominant scheduler for HPC and large training runs, has a worse security record than most ML engineers realize. It has historically run its daemons as root, and its authentication rests on `MUNGE` — a shared-secret credential system that was never designed for adversarial multi-tenant environments. [32] demonstrated this: attackers could bypass `MUNGE`'s hash-based message integrity protections in the `slurmd` process to replay root-level authentication tokens, gaining privileged access to compute nodes over the network. Earlier vulnerabilities were blunter — [33] let an unprivileged user send data to arbitrary Unix sockets on the host as root through `Slurm`'s `PMI2`/`PMIx` RPC handler, and [34] allowed privilege escalation to root via `SPANK` environment variable injection during Prolog/Epilog execution. `Slurm` clusters at national labs and cloud GPU providers routinely run versions months or years behind patches, because upgrading the scheduler means draining the entire cluster.

Ray's security posture is, by its maintainers' admission, a design choice rather than an oversight. Until recently, Ray had no authentication on its Jobs API or Dashboard — any process that could reach the network port could submit arbitrary code for execution on the cluster. Anyscale's position was that Ray should only run in isolated networks and act upon trusted code. The result was predictable: the ShadowRay campaign [35] compromised hundreds of thousands of exposed Ray clusters for cryptocurrency mining, data exfiltration, and eventually self-propagating botnets. Attackers extracted production database credentials, cloud environment tokens, and model weights from compromised AI workloads. The vulnerability remained unpatched for over two years because the vendor classified it as intended behavior.

Kubernetes introduces its own GPU-specific attack surface through device plugins. The NVIDIA device plugin runs as a privileged DaemonSet with access to the host's /var/lib/kubelet/device-plugins socket and device files. A compromised plugin can register fake GPU devices, intercept kubelet communications, or escalate to full node compromise. Kubernetes also lacks native GPU resource throttling — unlike CPU and memory, there are no cgroup controls for GPU compute time [36], so a malicious pod can monopolize GPU cycles and starve other tenants' inference workloads with no scheduler-level mitigation. In multi-tenant clusters, GPU memory is not isolated by default; processes on the same GPU can read each other's memory regions unless MIG (Multi-Instance GPU) or MPS partitioning is explicitly configured, and even then the isolation guarantees are weaker than what hypervisors provide for CPU and RAM.

### 2.3.2 Network Fabric

Large training runs communicate over InfiniBand or RoCE interconnects using RDMA — remote direct memory access that bypasses the kernel to move data between GPU memory regions at line rate. This is a performance architecture, not a security architecture. RDMA's kernel bypass means that the standard OS-level network monitoring and access control stack is not in the path. InfiniBand partition keys (P_Keys) provide coarse tenant isolation at the hardware level, but verifying that traffic isolation policies are correctly enforced across a fabric of thousands of ports is a manual, configuration-driven process with no runtime audit trail. The policy half of this problem has a formal foundation: NetKAT [37] gives forwarding policies a sound and complete equational theory in which reachability and tenant isolation are decidable equivalence checks. But its packet model targets OpenFlow-style Ethernet — nothing comparable exists for P_Key partitioning or RDMA fabrics, the gap that fabric policy verification (Section 3.9) addresses — and a verified policy is still a claim about what the controller was supposed to install. If you want to verify what is actually on the wire between nodes in a GPU cluster — not what the SDN controller *says* is on the wire — you need an independent observation point, which is the problem the verified network tap (Section 3.10) is designed to solve.

The SDN controllers that manage GPU cluster fabrics are themselves a concentrated target. An SDN controller is a single logical authority over all traffic routing decisions in the network; compromise it and you can redirect, mirror, or drop any flow. The OpenFlow protocol, which most SDN deployments use for controller-to-switch communication, did not require authentication of switches in early versions, and even post-1.2 versions with TLS support are frequently deployed without it [38]. An attacker who gains access to the control plane can add a second malicious controller (a feature available in OpenFlow 1.2+) and persistently reroute traffic without touching

the data plane hardware. The centralization that makes SDN manageable is exactly what makes it a single point of failure for traffic integrity.

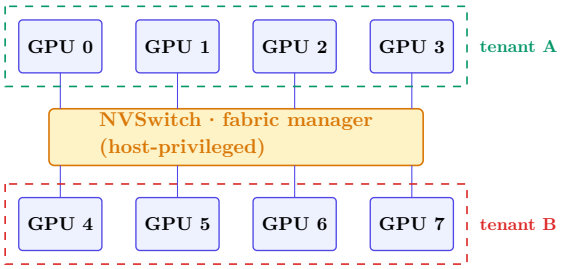


Figure 6: `NVLink`/`NVSwitch` topology inside a typical eight-GPU node. The fabric manager that programs `NVSwitch` routing tables runs as a privileged host process; the resulting tenant boundary is software-configured rather than hardware-enforced, and the traffic that crosses it never touches the CPU, the PCIe bus, or the OS network stack.

Within a single node, GPUs communicate over `NVLink` and `NVSwitch` at bandwidths up to 900 GB/s [39] — bypassing the host CPU, PCIe bus, and OS network stack. NVIDIA's fabric manager controls `NVSwitch` routing tables and restricts applications to designated address ranges, but this is a software-configured trust boundary, not a hardware-enforced one. There is no equivalent of IOMMU page-table isolation for `NVLink` traffic; the protection rests on correct configuration of the fabric manager, which runs as a privileged host process. A compromised fabric manager, or a bug in NVSwitch routing table setup, could allow one GPU's process to read or write another's memory region across the switch. For multi-tenant GPU servers using `NVLink`-connected GPUs — the standard configuration in `DGX` systems and cloud GPU instances — this is a lateral movement path that never touches the network and never appears in host OS logs.

Even when interconnect traffic is encrypted, traffic analysis remains viable. The volume and timing of gradient synchronization traffic are highly regular and predictable [40] — all-reduce operations produce characteristic burst patterns that correlate with batch boundaries, model architecture, and even convergence state. An observer on the fabric (or with access to switch port counters) can infer training progress, detect when checkpointing occurs, and distinguish between different model architectures without decrypting a single packet.

### 2.3.3 Distributed Systems

Distributed training coordinates gradient synchronization across hundreds or thousands of GPUs using collective operations — all-reduce, all-gather, reduce-scatter — implemented by libraries like `NCCL` and `Gloo`. These protocols run over the RDMA fabric with no authentication or encryption — NCCL's own documentation states it "does not provide secure network communication between GPUs" [41]; any process that can reach the network can inject or modify gradient data in transit. In federated and multi-node settings, this is the mechanism by which model poisoning attacks operate: a compromised node contributes manipulated gradients that shift the model's behavior while remaining within the statistical noise of normal training variance. The protocol's correctness properties — that all-reduce actually computes the sum of all participants' contributions, that no participant can observe another's individual gradient — are distributed-system invariants that formal methods can specify and verify. Proof-carrying code (Section 3.6) is one path to making these guarantees checkable at deployment time rather than assumed.

Checkpoint integrity is the other open wound. Large training runs write distributed checkpoints — sharded model state spread across dozens or hundreds of files on networked storage — every few thousand steps. These checkpoints are the recovery mechanism when nodes fail, which happens constantly at scale. They are also the artifact that gets promoted to production: the "final model" is just the last checkpoint that passed evaluation. Yet checkpoints are typically stored as serialized tensors with no cryptographic signature, no content-addressable hash chain linking them to the training run that produced them, and no tamper-evident log of which processes wrote which shards. An attacker with write access to the shared filesystem — or to the object store behind it — can modify checkpoint shards to inject backdoor behavior, and the training job will resume from the poisoned state without any indication that the checkpoint was altered. The same exposure applies when checkpoints are copied between clusters, uploaded to model registries, or handed off from a training team to a deployment team. The integrity gap between "this file exists on S3" and "this file is the authentic output of training run X at step Y" is currently bridged by convention, not cryptography.

Data pipeline integrity compounds the problem. Training data flows through distributed storage systems — `HDFS`, cloud object stores, data lakes — through preprocessing jobs, shuffling, and batching before it reaches the GPUs. Each stage is a point where data can be modified, filtered, or augmented by an attacker with access to the storage layer. Unlike gradient poisoning, which requires a compromised training node, data poisoning can happen long before training begins, in ETL jobs or data labeling pipelines that are often run with broader permissions and less monitoring than the training jobs themselves.

### 2.3.4 Identity and Access Management

The IAM layer controls which principals — human users, CI pipelines, serving infrastructure, and increasingly autonomous agents — can read model weights, write to training data stores, modify deployment configurations, or invoke models. The failure mode is over-permissioning: ML pipelines tend to accumulate broad service account credentials because the alternative is debugging permission errors during a training run. In 2025, `Sysdig`'s threat-research team documented an attack chain [42] where compromised credentials reached cloud administrator privileges in eight minutes across 19 distinct AWS principals — LLM-assisted reconnaissance followed by Lambda-based privilege escalation, with invocation logging disabled along the way. When the principal is an AI agent operating with long-lived API tokens, the IAM boundary *is* the control boundary. A token compromise gives the attacker everything the agent could do, with access patterns indistinguishable from legitimate automation. Proof-carrying code (Section 3.6) offers a structural alternative: rather than trusting that IAM policies are correctly configured and that tokens are uncompromised, require that actions carry machine-checkable proofs of authorization.

Model registries are an under-secured chokepoint. A model registry — `Hugging Face Hub`, a private `MLflow` instance, a cloud model catalog — is where trained models are stored, versioned, and promoted to production. Whoever can push a new version to the registry controls what runs in production. The PoisonGPT demonstration [43] showed that a surgically modified open-source model could be uploaded to `Hugging Face` and spread targeted misinformation while retaining normal benchmark performance. `ReversingLabs` found malicious code embedded in `pickle`-serialized models on `Hugging Face` that evaded the platform's safety scanning [19]. Model confusion attacks — the AI supply-chain analog of dependency confusion — exploit namespace reuse in

registries to trick downstream pipelines into pulling attacker-controlled models. Access control on model registries is often weaker than on code repositories: push permissions are granted to broad service accounts, there is no equivalent of signed commits or required code review, and the model artifact format (often `pickle` or `safetensors`) has no built-in integrity or provenance chain.

Secret sprawl is the background condition that makes all of the above worse. ML workflows generate and consume credentials at every stage: cloud storage keys in training scripts, `Weights & Biases` tokens in experiment configs, `Hugging Face` tokens in model download scripts, database credentials in data pipeline notebooks. `GitGuardian`'s 2025 report [44] found 23.8 million secrets leaked in public `GitHub` repositories, a 25% increase over the prior year — and ML repositories are disproportionately affected because the notebook-driven, experiment-oriented workflow encourages hardcoding credentials for quick iteration. These secrets leak into container images (baked into layers during build), into training logs (printed during debugging and never scrubbed), and into checkpoint metadata (serialized alongside model state). When an agent operates autonomously — writing code, calling APIs, launching training runs — it accumulates credentials in its context window and working files with no human in the loop to notice. The agent-to-agent delegation problem makes this worse: when one agent delegates a subtask to another, what credentials does the delegate inherit, and who revokes them when the task is done? Current IAM systems have no concept of transitive, time-bounded delegation for non-human principals.

## 2.4 Firmware and Low-Level Systems

*Related problems:* Microkernels and Hypervisors · Device Drivers for Verified Kernels · Weight Integrity and Kernel Supply Chain
*Invites:* Co-tenant · Rogue Insider · Supply Chain

Below the orchestration layer and above the silicon sits the firmware: hypervisors, device drivers, and boot chains. Code here runs at the highest privilege levels on both CPU and GPU. A bug is not a container escape — it is a host compromise, often with no log entry at all.

### 2.4.1 Microkernels and Hypervisors

Multi-tenant GPU isolation today relies on hypervisors, and the trusted computing base is enormous: `KVM`'s is roughly ten million lines of code [45], `Xen`'s smaller but still far past exhaustive verification. Both have had VM-escape vulnerabilities — Google Project Zero's 2021 `KVM` breakout via AMD `SVM` nested virtualization [46] is representative — and the PCI passthrough path that GPU workloads require widens the surface further, since IOMMU misconfigurations or missing PCI Access Control Services (ACS) can allow peer-to-peer DMA between devices assigned to different tenants. NVIDIA's Multi-Instance GPU (`MIG`) partitioning adds hardware-level memory isolation within a single GPU, but the partitioning is managed by the host driver, which runs inside the hypervisor's TCB. The verified-core response to a TCB that large — stop trusting most of it, verify a minimal core, and push policy and drivers into a de-privileged layer — is the microkernel line of work taken up as an enabler problem in Section 3.3 (`seL4`, `NOVA`, and the tailored hypervisors `seKVM` and AWS's Nitro Isolation Engine). None of those ships a verified driver for the accelerator itself, which is the widget at Section 3.4.

Even where `MIG` is deployed, GPU memory is not always scrubbed between context switches. Trail of Bits demonstrated this with LeftoverLocals [47]: on AMD, Apple, and Qualcomm GPUs, a co-resident process could read local memory left behind by a previous kernel, recovering roughly 5.5 MB per GPU invocation — enough to reconstruct an LLM's token-by-token output with

high fidelity. NVIDIA hardware was not affected in that specific case, but the deeper issue is architectural. GPUs were designed for throughput, not isolation, and the memory hierarchy reflects that priority. Shared L2 caches, shared interconnects, and performance counters that leak timing information all create side channels across tenant boundaries. The NVBleed attack [48] showed that contention on `NVLink` interconnects between GPUs lets an attacker fingerprint which deep-learning model a co-tenant is running with 97.8% accuracy, and the attack works across VM boundaries on Google Cloud — even after NVIDIA patched performance-counter access, the timing channel alone still yields F1 scores above 83%.

Confidential VMs compound the problem. CPU-side TEEs like AMD `SEV-SNP` encrypt guest memory so the hypervisor cannot read it, but extending that guarantee to a GPU is an open engineering problem. `SEV-SNP` requires the IOMMU in non-passthrough mode to prevent peripherals from reaching encrypted memory, which directly conflicts with the PCIe passthrough that GPU workloads need. AMD's `SEV-TIO` extension [49] is meant to bridge this gap using PCI-SIG's TDISP protocol, but it is not yet widely deployed, and the attestation story for a GPU behind a TDISP-secured link is still being worked out. In the meantime, any "confidential AI" deployment that claims `SEV-SNP` protection while using GPU passthrough has a gap in its threat model that the marketing materials do not mention.

### 2.4.2 Device Drivers and Runtimes

The GPU driver is the most privileged code that touches the accelerator. NVIDIA's proprietary `CUDA` driver stack, AMD's `ROCm`, and Intel's `oneAPI` are all closed-source, kernel-mode codebases that handle memory mapping, command submission, and context switching for every GPU workload on the machine. NVIDIA ships quarterly security bulletins; 2024 alone included [50] (privilege escalation in the display driver) and [51] (out-of-bounds read leading to code execution), plus nine vulnerabilities in the `CUDA` toolkit found by Palo Alto's Unit 42 [52]. These are not exotic attacks — they are standard memory-safety bugs in C code running at ring 0. The driver is a single point of compromise: an attacker who controls it can read any tenant's GPU memory, modify in-flight computations, or pivot to the host kernel. Because the driver source is proprietary, the only available mitigations are patching and hoping. A verified, open driver for at least the GPU command-submission path would change the calculus (Section 3.4).

A compromised or buggy GPU driver also opens the door to DMA attacks. The GPU sits on the PCIe bus and performs DMA to host memory; the IOMMU is supposed to restrict which physical pages the device can touch, but if the driver programs the IOMMU mappings incorrectly — or if an attacker can influence them — the GPU becomes a tool for reading or writing arbitrary host memory. [53] demonstrated exactly this on NVIDIA's Jetson platform, where a PCIe DMA attack bypassed secure boot. The IOMMU is a necessary defense, but it is only as good as the code that configures it, and that code is part of the same proprietary driver stack.

The `CUDA` compatibility layer adds its own surface area. NVIDIA maintains forward and backward compatibility across major `CUDA` versions, which means the driver must support multiple ABIs simultaneously and preserve behavior across a sprawling matrix of toolkit-to-driver version combinations. Operators running multi-framework training pipelines frequently pin older driver versions to avoid breaking their stack, which means known-patched vulnerabilities persist in production for

months or years. The compatibility shims themselves are additional code paths that rarely get the same scrutiny as the hot path, and any bug in ABI translation is a bug at ring 0.

On the open-source side, NVIDIA released its kernel-mode GPU modules as open source in 2022 [54], and the `Mesa` project's `NVK` driver now provides `Vulkan` support for `Kepler` through `Blackwell`. These are steps forward for auditability — community review can find classes of bugs that internal QA misses. But neither effort covers the full compute path that ML workloads use: the user-mode `CUDA` runtime, the compiler backend, and the firmware blobs that run on the GPU's internal microcontrollers all remain closed. An open kernel module with a closed firmware blob is better than nothing, but it is not a verifiable system. The gap between "source available" and "formally verified" is where the interesting work lies.

Worth naming what the open problem is and is not. The verified microkernels and hypervisors above (Section 2.4.1) all push device drivers out of the trusted core and into deprivileged user mode, which makes the driver verifiable in principle but does not make it verified. As a research methodology it works: separation-logic proofs of a driver against an abstract hardware model are a demonstrated pattern, from a `ZynqMP` DMA engine in concurrent separation logic [55] to BlueRock's `VirtIO` virtual switch [56], [57], [58]. The unsolved half is the abstract hardware model. The instruction set is the part that *is* modeled: Arm and RISC-V both ship machine-readable ISA specifications — Arm's even covers page-table walks — and REMS extends that line to relaxed virtual memory and the Arm MMU [59], [60]. What none of them covers is the accelerator's device-control surface: the GPU command processor, the IOMMU/SMMU programming interface, and the DMA engines. AMD publishes a machine-readable GPU ISA, but it is the shader instruction set, not that surface. The pivot from such a model to a software proof is a further gap; sketches exist [58] but no one has closed it at scale. The tractable problem at Section 3.4 is accordingly as much about producing a formal model of a device command interface as it is about proving a driver against one.

### 2.4.3 Boot Integrity and Firmware Supply Chain

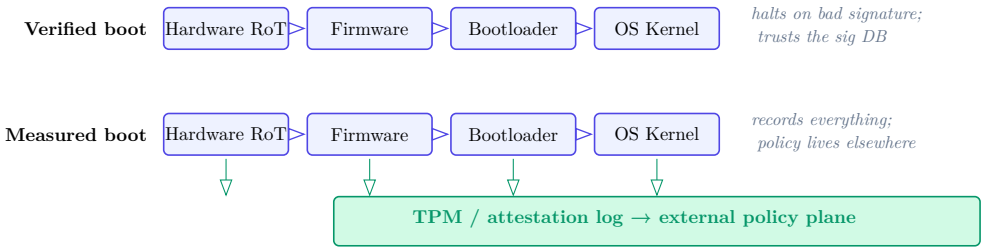


Figure 7: Two strategies for boot-chain integrity. *Verified boot* halts on a bad signature at each stage but trusts whatever lives in the signature database. *Measured boot* records hashes into a TPM and defers enforcement to an external policy plane that can release secrets, quarantine, or hard-reset based on attestation.

None of the isolation above matters if the firmware itself has been tampered with. Secure Boot and measured boot chains establish trust from the hardware root of trust (RoT) through each firmware stage to the OS kernel: each stage cryptographically verifies the next before handing off execution. NVIDIA's H100 extends this model to the GPU, with a per-device ECC keypair, on-die RoT, and a measured boot sequence that produces an attestation report — a signed manifest of every firmware component loaded. Combined with CPU-side TEEs (Intel `TDX`, AMD `SEV-SNP`),

this enables composite remote attestation: a verifier can check that both CPU and GPU booted clean firmware before releasing model weights or training data to a node.

There is a real architectural choice here, not just a terminology one, between enforcing boot locally via signature check (verified boot) and enforcing it externally via an attestation-gated control plane (measured boot). Verified boot halts on bad signatures, but it trusts whatever is in the signature database, and that database has been bypassed in the wild: the BlackLotus bootkit (2023) exploited [61] to defeat UEFI Secure Boot on fully patched Windows systems by bringing its own copy of a legitimately signed but vulnerable boot manager. Measured boot does not prevent anything from loading — the TPM faithfully records whatever hash is presented, clean or not — but in exchange it is more compositional: the hardware only measures, and policy (what to do with a given measurement) lives outside the boot path, in an operator-owned plane that can release secrets only to nodes that pass attestation, hard reset machines that fail it, or quarantine them from the management network for analysis. `NOVA` deliberately went this direction [7], on the argument that post-facto measurability with an external control plane is a better composition point than bundling enforcement into the boot ROM. Either story depends on continuous attestation and a fresh revocation list; a stale verifier leaves a tampered node operating undetected between checks.

The Baseboard Management Controller (BMC) is a separate, quieter threat. Every server in a training cluster has a BMC — a small system-on-chip running its own OS (often Linux), connected to its own network interface, with out-of-band access to the host's power, console, and firmware update mechanisms. BMCs are managed via `IPMI` or `Redfish`, and they are rarely patched with the same urgency as the host OS. In 2023, Binarly disclosed seven vulnerabilities in Supermicro BMC firmware [62] that gave unauthenticated attackers root access to the BMC. From there, an attacker can read cleartext credentials off the BMC filesystem, flash malicious UEFI firmware to the host, or pivot to every other BMC on the management VLAN. The persistence is the real problem: a BMC implant survives host OS reinstalls, disk wipes, and even GPU firmware updates, because it lives on a separate flash chip that the host never touches.

Firmware signing for GPUs is better than it was — NVIDIA's secure firmware update mechanism verifies digital signatures and enforces version anti-rollback — but the trust anchor is only as strong as the key management around it. The 2022 LAPSUS$ breach of NVIDIA [63] resulted in the theft of two code-signing certificates. Although the certificates were expired, Windows still accepts expired certificates for kernel drivers, and malware signed with the stolen keys appeared in the wild within a day. The broader lesson: a single key compromise turns the entire firmware-signing infrastructure from a defense into a distribution channel. What formal methods can contribute here is verifying the attestation protocol itself: proving that the chain of measurements is unforgeable, that a compromised node cannot replay a clean attestation, that revocation logic has no time-of-check/time-of-use gaps, and that the BMC's update path cannot be used to write outside its intended flash region. The machinery exists, but deploying it across a 10,000-GPU training cluster with continuous attestation, key rotation, and revocation checking is an engineering problem that remains largely unsolved at scale.

## 2.5 Hardware and Physical Supply Chain

*Related problems:* Device Drivers for Verified Kernels · Verified Network Tap
*Invites:* Supply Chain · Physical Adversary · Rogue Insider

At the bottom of the stack is the silicon itself: the GPU, TPU, and NPU dies that run every training and inference workload, and the global supply chain that produces them. Compromise here is durable: a hardware trojan survives every software update, every OS reinstall, every firmware reflash. It is also opaque: you cannot read out the RTL of a fabricated chip to check what it does.

### 2.5.1 Chip Architecture

The RTL design of a modern AI accelerator is proprietary. NVIDIA does not publish the register-transfer-level description of its GPU compute cores; neither do Google (TPUs), AMD, or any other vendor. This means the formal methods community cannot verify what the silicon actually computes. We take it on trust that an H100′s tensor cores implement matrix multiplication correctly, that its memory controllers enforce isolation between MIG partitions, and that no undocumented debug interfaces exist. Classen et al. [64] demonstrated that a hardware trojan implanted in a Xilinx `Vitis AI` DPU could backdoor a traffic-sign recognition model by replacing just 30 of 43,000 parameters at the accelerator level — expanding the circuit by 0.24% with zero runtime overhead, invisible to all software-side defenses. The attack needed no cooperation from the model or the training pipeline; the accelerator alone was sufficient.

The design tools themselves are part of the attack surface. The chip industry flows through a handful of EDA vendors — primarily `Synopsys` and `Cadence` — whose toolchains are millions of lines of proprietary code that no customer audits. Basu et al. demonstrated in their "CAD-Base" work [65] that every stage of the EDA pipeline — high-level synthesis, logic synthesis, place-and-route, verification — is a viable insertion point for hardware trojans. The place-and-route attack is nasty: the tool can emit one netlist for verification and a different netlist for tapeout, and existing equivalence-checking flows never compare the two. A malicious logic synthesis tool can exploit underspecified finite-state machines to insert extra transitions triggered by rare conditions, leading the circuit into attacker-controlled states. These are not hypothetical proof-of-concept attacks against toy designs; they target the actual tool interfaces that every fabricated chip passes through. Open-source EDA toolchains (`Yosys`, `OpenROAD`) exist but are not used for leading-edge AI accelerator design, so for now the industry trusts its toolchain vendors implicitly.

Beyond the design tools, modern SoCs integrate dozens of third-party IP cores — licensed blocks for PCIe controllers, memory interfaces, interconnect fabrics, security modules — that the chip designer incorporates without access to the underlying RTL. A third-party IP vendor can embed a hardware trojan that activates post-fabrication, leaks data through a covert channel, or degrades reliability on a trigger condition. The SoC integrator's only recourse is black-box testing and whatever contractual guarantees the IP license provides, neither of which catch a well-designed trojan. Formal verification of third-party IP at the pre-silicon stage is an active research area, but it requires a golden specification of correct behavior — which is exactly what you don't have when the IP is a licensed black box.

Hardware formal verification *is* used in industry, but its coverage is narrower than outsiders might expect. Intel adopted formal methods in earnest after the 1994 Pentium FDIV bug (a $475 million recall) [66]; by the Nehalem microarchitecture in 2008, formal verification was the primary validation method for floating-point and other arithmetic units. ARM's `ISA-Formal` framework [67] uses bounded model checking in `JasperGold` to verify that RTL implementations conform to the instruction set specification — but bounded model checking only proves correctness up to a

fixed sequence length, not for arbitrary execution traces, unless you can find the right invariants to lift it to an unbounded proof. In practice, formal verification at major vendors covers specific high-value blocks (FPUs, cache coherence protocols, security state machines) while the rest of the design relies on simulation-based testing. Nobody is formally verifying an entire GPU.

RISC-V changes part of this picture. The ISA specification is open, and several open-source RTL implementations exist (`BOOM`, `Rocket`, `CVA6`) with corresponding formal verification frameworks — `YosysHQ`'s `riscv-formal` [68] can check that an implementation conforms to the RISC-V spec using model checking, and the OpenHW Group's `CORE-V-VERIF` [69] provides industrial-grade verification for the CV32 and CV64 families. MIT's Riscy processors, built on the Coq-embedded `Kami` DSL [70], are formally verified open-source RISC-V cores. But open ISA is not open silicon. These designs still get fabricated at foundries through proprietary EDA toolchains and opaque manufacturing processes. An open RTL that you can verify pre-silicon is a real improvement over a proprietary RTL that you can't — but it does not close the gap between the verified netlist and the actual transistors on the die. FPGAs remain the one place where the deployer controls the entire path from RTL to running hardware, which is why Section 3.10 proposes FPGA taps as a verification target.

On the GPU side, the architecture determines what isolation primitives are available to driver software. Whether a verified driver (Section 3.4) can enforce strong tenant isolation depends on what the hardware exposes — and right now, we are relying on vendor documentation rather than verified specs for that interface.

### 2.5.2 Physical Security

Confidential computing has reached the GPU. NVIDIA's H100 supports a hardware-rooted TEE that, in conjunction with CPU-side enclaves (Intel `TDX`, AMD `SEV-SNP`), can encrypt model weights in transit and at rest on the device, with attestation that the firmware and driver stack have not been tampered with. This matters for AI because model weights are the most valuable artifact in the ecosystem — worth billions in training compute — and multi-tenant cloud deployments expose them to the cloud operator's entire software stack. But confidential computing protects against software adversaries with root access; it does not protect against physical adversaries with access to the hardware.

Modern AI accelerators use High Bandwidth Memory (HBM) stacked on a silicon interposer — the GPU die and HBM stacks sit side by side on a shared substrate connected by thousands of micrometer-scale copper traces. This interposer is a physical interface carrying model weights and activations at terabytes per second, unencrypted in current designs — even the H100′s confidential-computing mode draws the security boundary at the package and leaves on-package HBM traffic in the clear [71]. An attacker with physical access to the package could, in principle, probe these traces or tap the interposer to exfiltrate data. This is not a low-effort attack — it requires lab equipment and destructive or semi-destructive access to the package — but for a state-level adversary targeting a frontier model worth billions, it is not out of scope. HBM encryption at the interface level is not yet standard; until it is, the physical package boundary is the confidentiality boundary.

Side-channel attacks add another dimension. [72] showed that the magnetic flux from a GPU's power cable — measurable with a $3 induction sensor — betrays the detailed topology and

hyperparameters of a neural network being evaluated, achieving 96.8% classification accuracy for identifying network layers on an Nvidia Titan V. [73] demonstrated that electromagnetic emanations from GPU DVFS (dynamic voltage and frequency scaling) activity penetrate walls and can be captured at 3+ meters, enabling website fingerprinting and keystroke timing attacks. Scale these techniques to a data center with thousands of GPUs running a single training job, and the EM signature is correspondingly richer. TEMPEST-class shielding is well-understood for traditional compute environments, but GPU training clusters dissipate megawatts of power with correspondingly strong emanations, and it is not clear that standard data center construction provides adequate EM containment. This is an area where the AI security community has barely begun to assess the threat surface.

Counterfeit and recycled chips are a supply chain problem that predates AI but gains new significance when the chips in question carry model weights. The Semiconductor Industry Association estimates counterfeit semiconductors cost the U.S. industry roughly $7.5 billion a year in lost revenue [74]; the most common attack is remarking — taking a chip desoldered from e-waste, sanding the package markings, and relabeling it as a higher-grade or newer part. For commodity components this causes reliability failures; for security-critical components in a training cluster, it could mean a node with degraded or compromised silicon participating in a distributed training run. Detection methods exist — on-chip aging sensors ("odometers"), power side-channel profiling, X-ray inspection of die markings — but none are deployed systematically for GPU procurement at scale. The assumption is that buying from authorized distributors eliminates the problem, which is true until it isn't.

The geographic concentration of semiconductor manufacturing is a systemic risk that no amount of formal verification can address, but it shapes the threat model for everything else in this section. TSMC fabricates over 90% of the world's most advanced chips [75], including every NVIDIA AI accelerator. All of this capacity sits in Taiwan. A single earthquake, a fabrication contamination event, or a military conflict in the Taiwan Strait would halt frontier AI hardware production globally. The CHIPS Act and TSMC's fabs in Arizona, Japan, and Germany are slowly redistributing this concentration, but leading-edge capacity outside Taiwan remains years away from volume production. Advanced chip packaging — the 2.5D interposer and `CoWoS` technology that makes HBM-equipped AI accelerators possible — is even more concentrated than fabrication itself. For security planning, this means the physical supply chain has a single point of failure that cannot be engineered around with software or formal methods; it can only be mitigated through industrial policy and supply chain diversification.

For training clusters of 10,000+ GPUs, the physical supply chain itself becomes a target at the system level, not just the component level. Every node in the cluster must be attested before it receives model weights or gradient updates. Every network link between nodes carries data that, if intercepted or modified, could poison a training run or exfiltrate a model. Physical tamper-evidence for installed hardware — seals on server chassis, tamper-detecting rack enclosures, logged physical access — is standard practice in classified environments but not in commercial data centers running AI training workloads. As frontier model training costs reach hundreds of millions of dollars, the gap between the physical security posture of a commercial hyperscaler and the value of the assets it protects is widening. FPGA-based network taps (Section 3.10) address the link-level problem: a tap with a formally verified packet-processing pipeline can guarantee that it faithfully mirrors

traffic without injection or modification — a property that no software tap running on a general-purpose OS can credibly claim.

# 3 Tractable Problems

The problems below split into two kinds. **Enablers** are problems whose resolution unbottlenecks a class of downstream work. Adversarial robustness of formal methods (Section 3.1) is the clearest example: if an ITP's soundness can be broken by a sufficiently clever agent, then every verified artifact produced by that ITP inherits the vulnerability. Governance of neuralese proof stacks is similar — if proof oracles start emitting proofs in representations humans cannot audit, spec elicitation and validation become unsolved problems for everything else in this document. Solving an enabler does not itself produce a deployable artifact, but failing to solve it degrades the value of the widgets that depend on it. **Widgets** are concrete, scoped deliverables. Each widget stands on its own as a security improvement to a specific layer of the stack. Widgets are where the postdoc-years get spent and where the artifacts ship. The enabler/widget distinction matters for prioritization: an enabler with many dependents should be addressed early, and a widget whose value is contingent on an unsolved enabler should be sequenced accordingly.

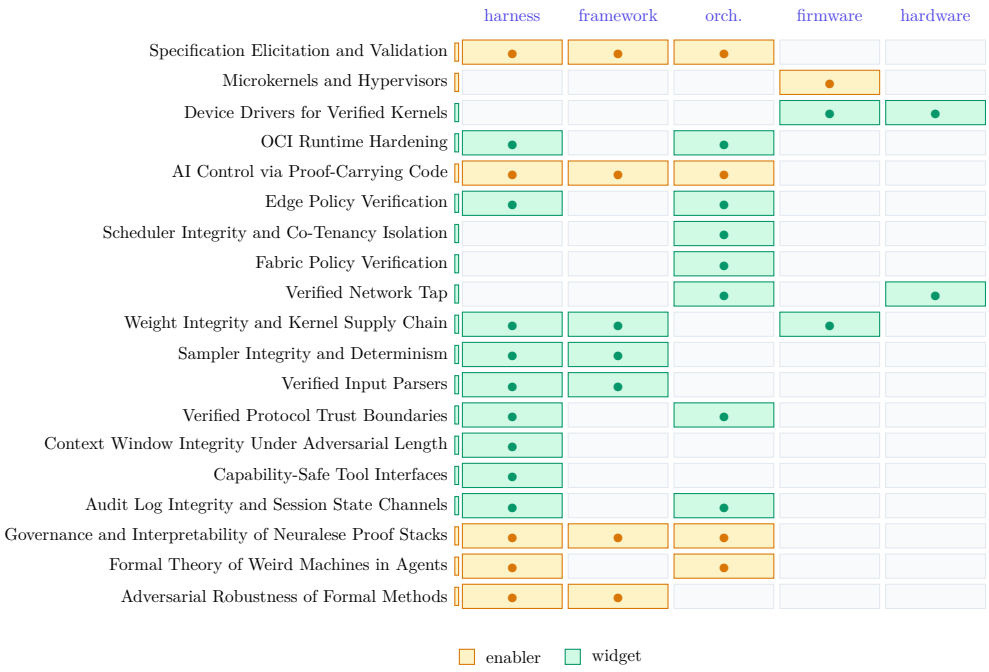


Figure 8: Tractable problems versus the stack layers they touch. Amber rows are enablers; green rows are widgets. A problem can touch more than one layer; most do. Section ordering matches the file include order in this chapter.

## 3.1 Adversarial Robustness of Formal Methods

*Stack layers:* Software & ML Framework | Execution Harness

*Blocks:* Malicious Model

`Agda`'s issue label "false" on GitHub sits at the time of this commit at 10 open and 76 closed. `Agda`'s issue label "false" tracks the *proofs of false* that Agda allows or has allowed. `Rocq` runs the same

play under a different name: the `critical` label is reserved for proofs of false. One asks, "isn't the whole point of a type theory that it be sound?"

So you see we have a problem. If ITP and other FM tools are not adversarially robust, scheming or reward hacking AIs will readily leverage novel zerodays to violate security properties. ARIA's *Safeguarded AI: Cybersecurity* solicitation [76] names a "verified proof-checking kernel" robust to adversarial AI proof generation as one of its example targets, framing the proof checker as the single point of trust if AI agents are to produce verified artefacts at scale — the same enabler, arrived at independently.

[77] discusses some of this to set up the `Lean` kernel arena. TODO: elaborate.

Broadly, soundness issues in proof assistants arise from the tension between expressivity and ease of use on one side and consistency on the other. Every concession to ergonomics — definitional extensions, universe polymorphism, coinduction, native compilation of the reduction machine, opaque conversion shortcuts — is a place where the kernel can drift away from the logic it was supposed to implement.

TODO: zero in on `Lean` specifically, (James Henson?), discuss governance challenges and the slippery definition of the problem.

### 3.1.1 Solution/project Sketch

Soundness bugs often arise due to the extreme complexity of a dependent type theory, who's proof checker we call a kernel ( 5k LoC in Lean, 10-30k LoC in Rocq). One idea would be to express the language of the ITP, even the dependently typed one, in a simpler theory with a smaller kernel. So you could, in principle, simply write a model of your complicated dependent ITP in a simple, well-trusted target and check proofs there.

What you want is a system in the LCF tradition (Milner's Logic of Computable Functions), whose defining discipline is that theorems are an abstract datatype and the only way to manufacture one is through a small fixed set of inference rules. That discipline is what makes the trusted core small, and it's preserved across systems with otherwise quite different logics: `HOL Light` [78] and `HOL Zero` [79] (classical higher-order logic), `Candle` [80] (HOL Light verified down through `CakeML`), and `Milawa` (a bootstrapped first-order prover in the `ACL2` lineage [81]). `HOL Zero` gets the trusted core down to a few hundred lines. Push further — verify the logic down to a large cardinal axiom, plus a model of the hardware, plus a small wrapper for actually running things — and rough estimates put the state of the art around 10k lines of spec.

One approach, then, is to pick something simple and well-trusted — ZFC or a variant, or a minimal LCF-style higher order logic, augmented with a large cardinal / large universe axiom — and reduce a working ITP all the way down to that. Useful prior art and reference points: `lean4lean` (a verifier for `Lean 4` written in `Lean 4`) and the `MetaRocq` (formalised metatheory of `Rocq` inside `Rocq`).

## 3.2 Specification Elicitation and Validation

*Stack layers:* Execution Harness | Software & ML Framework | Orchestration & Cloud

*Blocks:* Malicious Model

If proofs are cheap and specs are expensive, then specs are the bottleneck. Every widget in this document terminates in a spec: a verified sampler is verified against *some* statement of what a

sampler should do, an OCI runtime is hardened against *some* threat model, an IAM policy is checked against *some* notion of least privilege. The spec is where human intent meets machine-checkable formalism, and it is the part that no proof oracle can write for you — or rather, if a proof oracle writes it for you, you have quietly handed over the thing you were trying to keep.

Two subproblems:

- **Elicitation.** Getting the spec out of the stakeholder's head and into a formal language, without losing the parts they could not articulate. The classical software-engineering requirements-gathering literature is relevant but insufficient — we need elicitation techniques that produce artifacts a kernel can consume, not prose a PM can sign off on.
- **Validation.** Convincing yourself that the spec you wrote is the spec you meant. A proof of `P` against the wrong `P` is worse than useless: it launders a bug into a theorem. The property-based testing tradition [82] and the spec-mining tradition [83] are the two complementary handles we have on this — the first forces you to execute the spec against concrete inputs, the second infers candidate specs from observed behavior for the human to ratify or reject.

TODO: discuss the relationship to Section 3.18 — if the proof substrate is itself opaque, spec elicitation becomes unsolvable rather than merely hard, which is why we treat that as a separate enabler.

TODO: discuss differential specification (two independently elicited specs, checked for agreement) as a validation technique.

### 3.2.1 Solution/project Sketch

TODO: flesh out. Candidate directions: interactive spec-synthesis from examples, lightweight formal methods for spec sanity-checking, spec-level mutation testing, and human-factors work on what kinds of formal notation domain experts can actually read and write.

## 3.3 Microkernels and Hypervisors

*Stack layers:* Firmware & Low-Level Systems

*Blocks:* Co-tenant Rogue Insider

Multi-tenant GPU isolation today relies on hypervisors. While hypervisors started small, broadly they have grown into full cloud-operating systems. `KVM`'s trusted computing base is roughly ten million lines of code; and while `Xen` is smaller (roughly 400k LoC) it is still quite large. Both have had VM-escape vulnerabilities — Google Project Zero's 2021 `KVM` breakout via AMD `SVM` nested virtualization [46] is a representative example — and the PCI passthrough path that GPU workloads require widens the attack surface further, since IOMMU misconfigurations or missing PCI Access Control Services (ACS) can allow peer-to-peer DMA between devices assigned to different tenants. NVIDIA's Multi-Instance GPU (`MIG`) partitioning provides hardware-level memory isolation within a single GPU, but the partitioning is managed by the host driver, which runs inside the hypervisor's TCB.

Several projects have aimed to establish trust through verification at the hypervisor (and OS) level. To keep verification tractable, these projects aim to partition the system into a minimal trusted core (which holds exclusive access to page tables and privileged resources) and place policy decisions

in a de-privileged layer. This architecture reflects the principles of the microkernel philosophy and many of these systems are microkernels, or draw inspiration from that line of work.

seL4 [84] is the poster-child for micro-kernel verification. Popularized in the DARPA HACMS program [85], seL4′s proof connects a top-level specification written in Isabelle to a binary-level semantics of the underlying architecture. The depth of this proof allows it to remove trust from the C compiler (which is known to have bugs) through translation validation. seL4 supports a range of platforms and comes in different variants for different levels of isolation, including an MCS mode built for mixed criticality systems [86]. While seL4′s verification is deep, its proof for modern, server-class machines is somewhat limited. The proof is fundamentally sequential, meaning that the proof itself does not cover running seL4 across multiple cores. Recent work aims to address this through a multi-kernel configuration (where a separate version of seL4 runs on each core). In addition the sequential nature of the proof also assumes the absence of DMA by external devices, though practically speaking the underlying assumption really only requires the lack of DMA to seL4-controlled memory.

Sharing the microkernel-philosophy of seL4, the NOVA microkernel [7] aims to be a modern contender in the verified hypervisor space. In contrast to seL4, NOVA is highly aggressive with respect to concurrency and modern hardware adoption including features such as encrypted memory, measured boot, as well as IOMMU/System MMU support. While the verification of NOVA is still underway, the proof tackles the problem of concurrent verification from the onset leveraging a highly modular proof strategy and recent work (2026) has begun moving NOVA's proof to support the weak memory systems [87] present on modern architectures. In contrast to the seL4 proof, NOVA's (partial) proof bottoms out at the source code level leaving the potential for compilers to introduce bugs or vulnerabilities into the final binary.

Amazon's Nitro Isolation Engine [88] and the seKVM project [89] share similar goals in bringing verified system software to mainstream environments. Both projects verify a trusted core that runs in hypervisor mode and are driven by unverified but deprivileged components. Unlike seL4 and NOVA, these projects are not microkernels and their APIs are highly tailored to operate within their particular stacks. This narrower interface makes concurrency reasoning more tractable and both proofs support concurrent behaviors.

### 3.3.1 Beyond Hypervisor Mode

The architecture of isolating a small trustworthy (through verification) core from the higher-level requirements of modern hypervisors produces a good verification story, but the expressiveness of the API to the trusted module introduces a trade-off between capabilities and security. Projects such as seKVM embed policies within the verified core ensuring, e.g. that pages can not be shared (or read/write shared) between different domains. Embedding policies such as this directly within the verified code provides a strong guarantee, but can be limiting when trying to leverage these systems in contexts that seek more flexibility, e.g. allowing multiple domains the ability to communicate via shared memory.

On the microkernel side, seL4 and NOVA provide rich capability-based APIs that enable user-mode applications significant flexibility in resource allocation; however, the cost of this is that security guarantees such as isolation rely fundamentally on the correctness of the code that sets up these policies. seL4 provides the CAmkES framework [90] for configuring resources, but only supports

static configurations. NOVA's idiomatic usage is intended to be more dynamic; however, establishing isolation properties in this dynamic world requires user-mode verification. Efforts exist on both of these platforms to establish properties at the user-mode level. In the seL4 ecosystem, Kry10 [91] aims to raise the foundational guarantees of seL4 up the stack, and BlueRock Security has described how it has combined hypervisor- and user-mode verifications to build proofs of complete systems [92] (and technical reports on a verified VMM [57] and a verified virtual switch [56]).

### 3.3.2 Outlook and Next Steps

The "small trustworthy core" has been the crucial enabler for research and industrial systems to attack the hypervisor problem; however, the approach only goes so far. Neither seL4 nor NOVA, in isolation, is capable of running virtual machines, let alone supporting advanced features that many users expect from modern hypervisors such as dynamic provisioning and virtual machine migration. Bringing these core components into a trustworthy foundational stack is possible, but requires additional engineering effort to verify user-mode applications running atop these components. A core roadblock in this space is the sheer size of the technology stack that hypervisors support. Identifying a core feature set that would enable a wide swath of applications while still remaining tractable to verify with current technology is a key enabler for beginning to build these ecosystems.

A notable omission from the discussion above is device drivers. All of the hypervisor projects noted explicitly remove device drivers from the trusted core, a necessary step in controlling the size of the verified code; following the microkernel philosophy, device drivers are instead implemented in user-mode. While work on verifying device drivers exists [55], a major limiting factor is hardware complexity and modeling. While CPUs generally fall into only a handful of architectures, BlueRock's work on a verified network switch [56] provides some insight into the complexity of reasoning about a VirtIO device, but the lack of source code makes it difficult to fully evaluate. The accelerator-specific half of this problem — producing a machine-checkable model of a GPU's command and DMA surface and proving a driver against it — is the widget developed at Section 3.4.

### 3.3.3 Solution/project Sketch

Being an enabler rather than a single widget, the deliverable here is the *core feature set* itself: the smallest set of hypervisor capabilities that lets a verified core run a realistic multi-tenant GPU workload, scoped tightly enough to stay in reach of current proof technology. Start by working backwards from one concrete workload — a fixed set of inference tenants, each pinned to a MIG slice — and enumerate the capabilities it actually exercises: guest creation and teardown, second-stage page-table management, interrupt routing, and the IOMMU configuration that backs MIG isolation. Deliberately exclude what the deployment does not need yet (live migration, memory overcommit, dynamic device hotplug), recording each exclusion as an explicit assumption rather than a silent gap. The output of this first stage is a specification, not a proof: a machine-readable interface to the trusted core plus the isolation invariant it must preserve — *no guest can read, write, or fingerprint another guest's memory or its MIG-partitioned slice* — across every reachable sequence of capability calls.

Two build-out paths follow, and they can be pursued independently. On seL4, take the static-configuration story as far as it goes: express the workload's resource layout in CAmkES [90] and prove the isolation invariant for that fixed configuration, accepting the loss of dynamism as the

price of a closed proof. On NOVA, take the dynamic path: model the capability operations the workload issues at runtime and discharge the user-mode verification obligation that establishes the same invariant is preserved as guests are created and destroyed. Either path produces a reusable artifact — a verified minimal-hypervisor profile — and the gap between them measures exactly how much verification dynamism currently costs. The driver underneath both (Section 3.4) is the one capability this sketch assumes rather than builds; the longer arc is to fold dynamic provisioning and VM migration back in, one capability at a time, each addition re-discharging the invariant.

## 3.4 Device Drivers for Verified Kernels

*Stack layers:* Firmware & Low-Level Systems | Hardware & Physical Supply Chain

*Blocks:* Co-tenant | Rogue Insider

Multi-tenant GPU workloads today run on hypervisors whose TCB is orders of magnitude too large to verify, and whose device-facing drivers are proprietary kernel code running at ring 0 (Section 2.4.2). Every verified-core project — the seL4 [84] and NOVA [7] microkernels, and the tailored hypervisors seKVM [89] and AWS's Nitro Isolation Engine [88] — deliberately pushes device drivers *out* of the trusted core to keep the verified code small. The driver then runs deprivileged, in user mode, which is the right architecture but relocates the problem rather than removing it: tenant isolation now rests on the correctness of the user-mode driver and of the code that configures the kernel's protection policy.

The second dependency — the correctness of the code that configures the kernel's protection policy — is the capability-versus-policy trade-off taken up by the enabler problem at Section 3.3.1, where seKVM and Nitro bake policy into the verified core while seL4 and NOVA push it into configuration that must itself be verified. This widget targets the first dependency: the driver.

The research methodology for verifying a single driver — separation-logic proofs against an abstract hardware model, demonstrated for a ZynqMP DMA engine in concurrent separation logic [55], [57], [58] — is settled. The real blocker is that the abstract hardware model does not exist for the devices that matter. The CPU side is modeled — Arm and RISC-V ship machine-readable ISA specifications, and REMS reaches relaxed virtual memory and the Arm MMU [59], [60] — but no GPU vendor publishes a machine-checkable specification of its command processor, on-GPU memory-management and IOMMU interfaces, or DMA engines. AMD's machine-readable GPU ISA is the shader instruction set, not that control surface. So the tractable problem is two-sided: produce a formal model of a device interface at a fidelity that supports proof, and prove a driver against it. The longer prize behind it is a verifiable *feature set* — enough of a hypervisor (dynamic provisioning, VM migration) and its user-mode drivers to run real workloads, kept small enough to stay in reach of current proof technology. Neither seL4 nor NOVA can run a virtual machine in isolation today; identifying that minimal feature set is the precondition for everything above it.

### 3.4.1 Solution/project Sketch

Start with the smallest useful surface: the command-submission path of a single open-source GPU stack (NVK/Nouveau on NVIDIA, or an AMDGPU subset). Specify the command-ring state machine in Rocq or Lean at a level of detail that admits noninterference claims across tenant contexts — ring-buffer consistency, IOMMU mapping integrity, context-switch scrubbing. Prove a reference driver (runnable under either seL4′s sDDF or NOVA's userspace driver model) against that spec, with the security property being *no sequence of guest-supplied command packets causes the driver*

*to program an IOMMU mapping or issue a DMA outside the guest's declared memory region.* Two natural stopping points: a verified command-submission module with a stubbed-in hardware model, which is shippable on its own as a reference; and the same driver proved against a model co-developed with the vendor or with REMS, which is the research contribution.

## 3.5 OCI Runtime Hardening

*Stack layers:* Orchestration & Cloud | Execution Harness

*Blocks:* Malicious Model | Co-tenant

The standard container runtime, `runc`, implements the OCI specification atop Linux namespaces and cgroups. It was never designed to resist an adversary *inside* the container, and the CVE record reflects this. When the workload in question is an AI agent with tool use (i.e. one that can write and execute arbitrary code), the gap between "process isolation" and "security boundary" becomes the whole game.

Recent benchmark work has started to quantify the gap. [93] measures frontier LLM capabilities for container sandbox escape directly, treating the model as a red-team operator. The UK AISI sandboxing toolkit [94] and its Docker Sandbox Breakout CTF [95] provide standardized escape scenarios built on `Inspect AI` [96]. [97] introduces BashArena, a control setting specifically for highly privileged agents. BoxArena [98] inverts the usual offensive framing: instead of asking "can this agent escape?", it fixes the attacker model and varies the OCI runtime, producing a *defensive* leaderboard. Five attack surfaces (filesystem, socket, process, network, syscall) are tested across runtimes like `runc`, `gVisor`'s `runsc`, `crun`, and `Kata Containers`.

The pattern across all of these is convergent: empirical red-teaming finds holes, patches get applied, and next quarter a new escape surfaces. A sufficiently capable agent should be assumed to find any flaw in its containment. This is where formal methods enters.

### 3.5.1 Solution/project Sketch

What would a *formally verified OCI runtime* look like? The lineage runs through the `CLInc` stack [99] (end-to-end verified from gate-level hardware through compiler to application) and `seL4` [84] (functional correctness of a microkernel, machine-checked in `Isabelle/HOL`). A verified OCI runtime would prove that the namespace and capability isolation enforced by the runtime matches a formal policy – that no sequence of syscalls from inside the container can violate the confinement invariant.

This is a large artifact, but it decomposes into tractable pieces: a formal model of the relevant Linux ABI surface (the  80 syscalls a hardened container actually needs), a specification of the confinement policy (drop all capabilities, read-only rootfs, `seccomp` allowlist, no-new-privileges), and a proof that the runtime's state machine preserves the policy across all reachable states.

The AI control literature [100], [101] adds another dimension: even with a verified runtime, you want a *monitor* – a second model or policy that reviews agent actions before they reach the kernel. The combination of a verified isolation boundary and a scalable monitor protocol is stronger than either alone, and the monitor's job gets easier when the boundary it sits behind has a proof.

## 3.6 AI Control via Proof-Carrying Code

*Stack layers:* Execution Harness | Software & ML Framework | Orchestration & Cloud

*Blocks:* Malicious Model

Proof-carrying code [102] and AI Control [100], [101] are two literatures about the same problem: keep an untrusted producer on the far side of an interface and let only a small trusted checker decide what gets through. PCC's answer is that every proposed action arrives tagged with a machine-checkable certificate; the trusted base is the proof kernel and nothing else. AI Control's answer is that a weaker trusted monitor, backed by a scarce audit budget, decides per-action on a safety–usefulness Pareto frontier. The deterministic midpoint — the untrusted agent emits code plus a proof in a standard ITP, the boundary rejects anything that does not typecheck, and the trusted base collapses to the kernel plus a human-written specification — has been sketched at prototype scale [103] but not stress-tested. ARIA's *Safeguarded AI: Cybersecurity* solicitation [76] names "verified output mediation for AI inference" as an example target: a proof that every externally visible output — token stream, tool call, actuation request — traverses the monitor chain and that no bypass path exists. That is the structural (complete-mediation, non-bypass) half of this enabler, stated as a reference-monitor property rather than a proof-carrying one; the two compose, since a non-bypassable boundary is where the typecheck-or-reject gate has to sit.

The question this enabler addresses is whether that synthesis can actually pay for itself. Classical PCC was designed for a world where proofs were written by compiler-level procedures on statically-known properties. Extending it to a world where the code producer is a frontier model and the safety property is "this action does not compromise the serving stack" raises a quantitative issue: the *verification burden* $k$ — the ratio of proof-synthesis cost to program-synthesis cost. A world where $k$ is small is a world where AI-control-via-PCC is the right architecture for most widgets in this document. A world where $k$ is large — or where it is small only for spec-shapes nobody writes in practice — is a world where those widgets have to be verified individually, outside any unified control protocol.

Three open subproblems determine where $k$ lands.

*Proofs as a matter of degree.* Current type checkers give a binary signal: the proof closes or it does not. A partial proof that is ninety percent of the way to closure is indistinguishable, to the kernel, from a partial proof that is ten percent there. A control protocol that could read *how close* a submitted proof is to closing — via a learned value function over tactic sequences, a conditional-probability estimate on the remaining subgoal structure, or a simpler admissibility heuristic — could admit "warm" proofs under bounded risk and reject "cold" ones as likely unprovable. This replaces the all-or-nothing gatekeeper with a graded one, and the grading itself becomes a research target.

*Specialist blue-team models.* Nothing in the AI Control framing requires the trusted checker to be a general model. A small model fine-tuned on a specific Hoare fragment, with custom tactics targeting a specific surface (context-window bookkeeping, permission-attenuation lemmas, IFC label discharge), exploits the asymmetry that verification is sometimes provably cheaper than generation. The training cost is upfront and amortized over deployment; the deployment cost is low by construction. The open question is how much of the security-relevant spec space admits specialists this way.

*Upfront versus ongoing investment.* The seL4 project [84] spent roughly 2.2 person-years on kernel design and coding, 9.0 on reusable infrastructure and tooling, and 11.0 on kernel-specific proof engineering — roughly a $1:1$ ratio of better-swords to more-fencing. In a PCC-for-AI world, who

pays for the tooling half? If the red team pays — compliance as a cost of doing business, on the SOC-2 model — the original Ctrl constraint that *blueteam_budget* much-less-than *redteam_budget* dissolves, and the verification-burden arithmetic changes sign. If the blue team pays, the constraint binds and the first two subproblems become load-bearing.

### 3.6.1 Solution/project Sketch

Four threads, sequenced by how much they unblock.

*Verification-burden measurement at scale.* Measure $k$ on the spec shapes that actually matter for this document: context-window allocation (Section 3.15), sampler conformance (Section 3.12), tool-composition safety (Section 3.16, Section 3.19), and weight-load refinement (Section 3.11). Produce $k$ distributions per spec class, per model, per year. This is an ongoing benchmark, not a one-shot paper.

*Warmness metrics and graded gatekeeping.* Build partial-proof signals into a control protocol. Start with `Lean` tactic-sequence value functions trained on human and synthetic proof traces, extend to subgoal-structure embeddings, and evaluate as a replacement for binary pass/fail in a backdoored-code testbed in the style of [100]. The deliverable is a graded gatekeeper with a published ROC curve on audit-budget-vs-catastrophe-rate.

*Specialist-checker training.* Pick one narrow spec surface — say, Hoare logic over the context-assembly allocator — and train a small model specifically for its proof obligations with custom tactics. Measure the specialist's $k$ against a general model's $k$. Repeat across surfaces until the specialist-viability boundary is clear.

*Control protocols as a programming language.* Speculative and out-of-scope for the near-term, but worth naming. The Dolev-Yao model [104] reduced cryptographic-protocol reasoning to logic programming and absorbed a class of attacks into definitional unreachability. An AI-control analogue — a process calculus in which resampling, deferral, and trusted-editing primitives have first-class semantics and attacker capabilities are syntactic — would let protocol designers discharge parts of the soundness argument at design time rather than per deployment. PLT or FV researchers looking for an entry point into AI safety should consider this direction.

## 3.7 Edge Policy Verification

*Stack layers:* Execution Harness Orchestration & Cloud

*Blocks:* External User Malicious Model

The API gateway in front of an inference service decides two things on every request: whether the caller is who they claim to be, and whether they have exceeded their allowed rate. Both reduce to the same kind of property — over any window of length $T$, principal $X$ may send at most $N$ requests — with authentication setting which $X$ applies and rate limiting enforcing the bound. They also fail the same way: the protocol's state space is large enough that production implementations exhibit reachable states no one designed for. JWT algorithm confusion [105] let attackers forge tokens by switching from RSA to HMAC verification, a bug that lived in production libraries for years because the auth handshake's state space was never systematically explored. OAuth 2.0 compounds the problem: authorization codes, refresh tokens, PKCE challenges, and session binding interact in enough combinations that implementation errors routinely produce exploitable states. On the quota side, distributed counters (`Redis`, or in-process sliding windows

with gossip) admit concurrent-write race conditions that silently let burst traffic exceed policy — the system reports correct enforcement while the count drifts above the limit. The standard response to both failure modes — unit tests and penetration testing — samples the state space rather than covering it.

Authorization for agent-originated requests is a different shape of problem. In agentic deployments, the model generates tool calls — HTTP requests, database queries, shell commands — that carry the *human* session's credentials. The model acts on behalf of the user but is not the user; it is a textbook confused deputy. If the routing logic does not attenuate authority, a prompt injection or jailbreak that causes the model to emit an unexpected tool call inherits whatever permissions the session holds. The question here is not "are the right principals allowed through" but "is the set of actions reachable through model-generated requests a strict subset of what the human intended to authorize" — a containment property over a capability graph, not a rate property over a credential state machine.

The two concerns share an architectural site (the gateway) but separate cleanly as formal targets, and they are treated below as two distinct projects.

### 3.7.1 Solution Sketch A: Rate-Bounded Authorization

Build a `TLA+` model of the gateway's credentialed-request lifecycle as a single state machine: authentication maps a request to a principal, then a per-principal counter decides whether to admit. On the authentication half, verify that no reachable state admits a token under an unintended algorithm, an expired credential, or a forged signature — covers JWT algorithm confusion and the OAuth state-machine errors above. Extend with a `ProVerif` [106] or `Tamarin` [107] model of the full OAuth 2.0 flow, producing machine-checked proofs of token secrecy and session binding. On the rate-limit half, express the policy as a `TLA+` invariant — "over any window of length $T$, the count for principal $k$ does not exceed $N$" — and verify the distributed counter implementation against it under all process interleavings and crash-recovery scenarios. Where the counter uses shared mutable state, apply `Iris` or another concurrent separation logic to prove the monotonicity invariant (the counter never decreases except at window expiry). The two halves share the principal identifier as their interface: the auth model guarantees principal identity is sound, and the rate-limit model guarantees enforcement is correct given that identity. The deliverable is a verified reference gateway with both halves discharged in the same `TLA+` model file, plus a test harness that regression-checks deployed gateways against the spec.

### 3.7.2 Solution Sketch B: Agent Authority Attenuation

Model the gateway's agent-mediation logic as a capability system in `Alloy`: the human session holds a capability set, the model's tool-call interface holds an attenuated subset, and the routing logic mediates between them. The specification: no sequence of model-generated requests can reach a capability outside the attenuated set — a direct confused-deputy analysis of the gateway's forwarding rules. The vocabulary of capabilities — what they name, what authority they convey, how they attenuate for tool calls — is exactly what Section 3.16 develops; this sketch consumes that vocabulary and verifies that the gateway's enforcement of it has no escape paths. The deliverable is a verified reference for the gateway's tool-call routing layer (or shim), with proofs that the attenuation function is complete (every model-reachable action lies inside the attenuated set) and that the routing rules are sound with respect to it.

## 3.8 Scheduler Integrity and Co-Tenancy Isolation

*Stack layers:* Orchestration & Cloud

*Blocks:* Co-tenant Rogue Insider

The dominant real-world mitigation for co-tenant attacks is not a scheduler policy but a purchase order: hyperscaler customers with serious confidentiality requirements buy reserved or bare-metal capacity and opt out of shared tenancy entirely. This section is scoped to the deployment model where that option is not on the table — mid-tier cloud providers, academic and national-lab clusters, and internal multi-team platforms at organizations without per-team hardware budgets. In that setting, the cluster scheduler (`Kubernetes`, `Slurm`, or a custom allocator) decides which physical host runs which job, and the correctness of that decision is load-bearing for any isolation claim the operator wants to make.

The framing matters because the question "which tenant pairs are adversaries" is not answerable a priori. An attacker knows their target; the operator, ahead of time, does not. So the tractable formal problem is not "identify dangerous co-locations" but "given an isolation policy the operator has specified, prove the scheduler enforces it". The placement logic is a constraint solver operating over dozens of dimensions (resource requests, affinity, taints, topology spread, GPU fabric topology), and the interaction between constraints is hard to reason about by inspection. A misconfigured or under-constrained policy can silently permit co-location the operator assumed was forbidden — and the failure is detectable only post-factum, by which point the attacker has already been a co-tenant.

Resource accounting has none of this scoping difficulty. Attribution fraud (one tenant's GPU-hours billed to another) and quota evasion (a malicious job consuming more than its allocated share without detection) apply to every shared-tenancy deployment regardless of whether anyone is trying to side-channel anyone else. The metering pipeline — usage counters, aggregation, attribution across concurrent job start/stop and preemption — is a distributed accounting protocol with the same class of race conditions that make distributed rate limiting hard, and it admits the same class of formal treatment.

The residual confidentiality story — cache side channels (Flush+Reload [108], Prime+Probe [109]) against a co-resident victim — then becomes a secondary application of the same placement model: given a formally enforced policy, one invariant of that policy can be "no two jobs labeled *mutually-distrusting* share an LLC domain or GPU memory fabric", and the same verification discharges it.

### 3.8.1 Solution/project Sketch

Two workstreams, both grounded in policy-enforcement rather than threat identification.

First, accounting correctness. Model the metering pipeline (counters, aggregation, attribution) as a distributed state machine and prove with concurrent separation logic (`Iris` or equivalent) that the accounting invariant holds: every GPU-second is attributed to exactly one tenant, and the sum of attributed usage equals actual usage, under concurrent job start/stop, preemption, and node failure. This is universally applicable and does not depend on any adversary-identification assumption. The deliverable is a verified reference metering library with a concurrency-safe aggregation protocol, usable as a drop-in for existing schedulers.

Second, placement-policy enforcement. Model the cluster topology (hosts, LLC domains, GPU memory fabrics, network segments) and the operator-authored isolation policy (expressed as relational constraints over tenant labels and topology resources) in `Alloy` [110]. The verification question is not "is this policy the right policy" — that is a business decision the operator owns — but "does the scheduler's constraint solver, across all feasible cluster states and job arrival sequences, only produce placements that satisfy the stated policy". Counterexample traces from bounded model checking become a conformance test suite against a concrete scheduler (e.g., `Kubernetes` with a given set of affinity and taint rules). The deliverable is a policy-specification language, an `Alloy` model of a representative scheduler, and an audit shim that checks live placement decisions against the model.

## 3.9 Fabric Policy Verification

*Stack layers:* Orchestration & Cloud

*Blocks:* Co-tenant Network MITM

Every tenant-isolation claim on a GPU cluster fabric bottoms out in configuration state: `P_Key` partition tables and forwarding tables programmed by the `InfiniBand` subnet manager, `NVSwitch` routing tables programmed by NVIDIA's fabric manager, VLAN and ACL configuration on the `RoCE`/Ethernet side (Section 2.3.2). None of these artifacts has a semantics. The operator's evidence that tenants A and B cannot exchange packets is that someone read the config and believes it; across a fabric of thousands of ports under continuous reconfiguration — node failures, topology changes, jobs arriving and departing — that belief is not auditable, and a single mis-programmed table entry silently violates the isolation policy without violating anything the fabric itself checks.

For OpenFlow-style networks this problem is solved in principle and increasingly in practice. NetKAT [37] gives forwarding policies a sound and complete equational theory in which reachability and slice isolation are equivalence checks, and KATch [111] decides those checks symbolically on realistic topologies in under a second — fast enough to re-verify on every configuration change rather than once at design time. What does not exist is the packet and forwarding model for the fabrics AI clusters actually run on. NetKAT's semantics speaks OpenFlow Ethernet: match on header fields, modify, forward. `InfiniBand` subnet-managed routing, partition-key enforcement at ports and channel adapters, and `NVSwitch` address-range routing are different enough that the existing tooling does not apply off the shelf — but they are not different in kind. The bet this widget makes is that the KAT machinery transfers and the missing work is modeling, not new theory.

### 3.9.1 Solution/project Sketch

The deliverable is a NetKAT-style semantics for RDMA fabric configuration, a decision procedure for tenant non-interference over it, and an audit shim that runs the check continuously against live fabric state.

First, the semantics. Model the `InfiniBand` subnet as a KAT policy: linear forwarding tables become the forwarding relation, `P_Key` membership tables become tests at ingress and egress ports, and the subnet manager's view of the topology becomes the network term. Tenant isolation is then the standard NetKAT non-interference query — the slice restricted to tenant A's `P_Key` composed with the slice restricted to tenant B's is equivalent to `drop` — decided by an equivalence check in the style of [111]. The `NVSwitch` case is the same shape one level down: the fabric manager's

routing tables and address-range restrictions are the policy, and the isolation question is whether any GPU in one tenant's partition can reach an address range belonging to another's.

Second, the audit shim. The verified artifact is worth little if it checks a config that drifted from the fabric hours ago. Extract live state — `OpenSM` forwarding and partition tables, fabric manager routing state — compile it into the model, and re-run the non-interference check on every reconfiguration event, with a signed log of check results as the audit trail. This is the policy-side complement of the verified network tap (Section 3.10): the shim proves the installed configuration enforces isolation, and the tap's monitor stream is the evidence the fabric behaved as that configuration prescribes. It also gives the scheduler co-tenancy model (Section 3.8) something to discharge against — a placement policy's "no shared network segment" constraint becomes a claim checkable in the fabric model rather than an assumption about labels.

Two natural stopping points. A `P_Key`-level model checked against real `OpenSM` table dumps from a production cluster, with the non-interference query running in seconds, is shippable and immediately useful to any operator making isolation claims to customers or auditors. The research contribution is the dynamic case: modern fabrics use adaptive routing, which reroutes traffic around congestion at runtime, and whether the isolation proof survives the full reachable set of adaptive-routing states — rather than the single static table the subnet manager last wrote — is exactly the kind of question the equational theory exists to answer and nobody has posed to it.

## 3.10 Verified Network Tap

*Stack layers:* Orchestration & Cloud | Hardware & Physical Supply Chain

*Blocks:* Network MITM | Physical Adversary

The control plane that decides what flows where on a GPU cluster fabric — SDN controllers, fabric managers, tenant routing tables (Section 2.3.2) — is the same piece of infrastructure whose compromise is the threat. An operator's own logs about what crossed which link cannot be evidence in a threat model where the operator's logging stack is on the attacker's side of the trust boundary. Independent observation on the fiber itself is the difference between "the controller says traffic between tenants A and B was partitioned" and "no such packet appeared on the inter-pod link during the audit window." The tap composes with policy-level verification (Section 3.9) rather than replacing it: NetKAT [37] makes tenant isolation a decidable property of the *intended* forwarding policy, and the monitor stream is what lets an auditor check that the fabric actually behaved as that policy prescribes. Without the former, conformance is checked against an informal spec; without the latter, the proof is about a configuration the compromised controller may never have installed.

[112] develop this as the verification primitive for AI-datacenter compliance under mutual distrust between operator and external verifier. Their cost figures are the encouraging part of the picture: a north-south tap at the datacenter edge captures all external traffic at well under 0.01% of facility upfront cost; an east-west storage-fabric tap with full hash-and-timestamp capture sits at 0.3-0.75%; compute-fabric sampling via optical circuit switches at 0.2-1.5%. FPGA capture at 400 Gbps and above already has precedent in financial-regulation hardware (Arista 7130, Deutsche Börse). [113] adds the optical reality the spec has to live with: passive splitters add no latency but eat link budget — a 1 dB loss can break a 53 Gbaud link — while active OEO taps regenerate the signal at roughly 20 ns latency and tens of watts but introduce a new failure point on the live

link. The same artifact serves an internal threat model (compromised SDN controller, dishonest fabric manager) and an external one (treaty verifier, regulatory audit, third-party red team); the LTL contract below is identical, and the threat-model split surfaces only in key custody and in who is allowed to read the monitor leg — policy rather than mechanism (Section 1.3.1).

### 3.10.1 Solution/project Sketch

The deliverable is the LTL contract below plus an open-source RTL reference design proven against it, for an active optical-electrical-optical tap at 400 Gbps with a path to 800 Gbps. FPGA is the expected deployment target: [112]'s 400 Gbps capture precedent (Arista 7130, Deutsche Börse) is FPGA-based, and the audit story benefits from a re-flashable bitstream tied to a published proof. But the contract is platform-agnostic — an ASIC implementation that meets P1-P6 is equally valid, and an open-source RTL reference is what gets verified, not any particular bitstream. Three pieces: a hardware specification in `Cryptol` or `SystemVerilog` with proofs in `SymbiYosys` or `Kami` against the contract below; a tamper-evident reference design with externally inspectable interfaces; and an out-of-band verification cluster that consumes the monitor stream and runs compliance checks asynchronously, so the wire-side hardware stays simple enough to verify while sophisticated analysis happens elsewhere.

Let `live_in(p, t)` mean packet `p` was clocked into the tap on the live ingress at time `t`; `live_out(p, t)` that it was clocked out on the live egress; `mon(r, t)` that the monitor leg emitted a record `r` at time `t`; and `fail(t)` that the tap has self-reported a fault. Let `D` and `D'` be the forwarding and mirroring deadlines in clock cycles, and let `image(p)` be the deterministic projection of `p` into the monitor record format (header parse + timestamp + payload hash, per [112]).

Using $\Box$ for "always", $\Diamond$ for "eventually", $\blacklozenge$ for the past-time "once", and $\Diamond_{\le D}$ for "eventually within $D$ steps":

$$\Box(\text{live\_in}(p) \rightarrow \Diamond_{\le D}\ \text{live\_out}(p)) \qquad \text{(P1) forwarding completeness}$$

$$\Box(\text{live\_out}(p) \rightarrow \blacklozenge\ \text{live\_in}(p)) \qquad \text{(P2) forwarding fidelity}$$

$$\Box(\text{live\_in}(p) \rightarrow \Diamond_{\le D'}\ \text{mon}(\text{image}(p))) \qquad \text{(P3) mirror completeness}$$

$$\Box(\text{mon}(r) \rightarrow \blacklozenge(\exists p.\ \text{live\_in}(p) \wedge \text{image}(p) = r)) \qquad \text{(P4) mirror soundness}$$

$$\Box(\text{live\_out}(p) \rightarrow \neg\ \text{depends-on}(p, \text{mon-input})) \qquad \text{(P5) no back-channel}$$

$$\Box(\text{fail} \rightarrow \Box(\text{live\_in}(p) \rightarrow \Diamond_{\le D}\ \text{live\_out}(p))) \qquad \text{(P6) fail-to-wire}$$

P1 and P2 together discharge the transparency obligation against the live forwarding path: nothing dropped, nothing synthesized. P3 and P4 discharge the monitor-leg faithfulness obligation: no missing packets, no fabricated ones. P5 is a hyperproperty rather than a pure LTL formula — a noninterference statement that the live-out signal does not depend on any input received via the monitor leg — and is the property that defeats the verifier-collusion attack in [112]'s threat model. P6 is the fail-open invariant from [113]: a tap component failure must downgrade the device to a transparent pass-through, not a link-cut, because the live link is load-bearing for production traffic.

The projection `image(p)` is where the spec interacts with policy. For east-west compute-fabric taps, `image(p)` is a header parse plus a SHA-3 payload hash — the verifier learns timing and routing without payload contents. For north-south edge taps, the operator may retain session keys for

post-hoc decryption under audit. Both factor through the same LTL contract; only the definition of `image` changes, which is the spec-implementation separability principle from Section 1.3.3.

Two natural stopping points. A verified single-port tap with a stub fail-detection path, proven against P1-P6 modulo a hand-modeled SerDes layer, is shippable as a reference design. The same tap proven against a co-developed model of a real 400G PCS/SerDes is the research contribution; this is the same hardware-model gap that blocks GPU driver verification (Section 2.4.2). Multimode fiber is out of scope — the link budget does not admit passive splitting at high baud rates [113] — and the industry trend toward single-mode-only AI fabrics is the precondition this widget ships against.

## 3.11 Weight Integrity and Kernel Supply Chain

*Stack layers:* Execution Harness | Software & ML Framework | Firmware & Low-Level Systems

*Blocks:* Supply Chain | Rogue Insider

Model weights are the highest-value static artifact in an ML deployment. A frontier model's checkpoint runs tens to hundreds of gigabytes, sharded across a distributed filesystem, and any bit-flip or deliberate modification in those shards changes the model's behavior in ways that are hard to detect at inference time. The threat model is straightforward: an adversary with write access to the storage layer — compromised NFS node, supply-chain attack on a model registry like `Hugging Face`, or a privilege escalation on the serving host — rewrites a weight shard. Existing defenses amount to hash-checking at load time. But the loader itself is a large, unverified C++/Python codebase, and the check is only as trustworthy as the code that performs it. A compromised loader can skip the check, and a hash collision (unlikely but not formally excluded) defeats it silently.

A second, orthogonal problem lives one layer deeper. The compute kernels that execute the forward pass — matrix multiplications, attention, activation functions — are supplied by hardware vendors as precompiled binaries. Nvidia's `cuBLAS` and `cuDNN` are closed source. AMD's `MIOpen` is partially open. In both cases, the optimized kernel that actually runs on the GPU is not the code anyone audited. A compromised kernel could introduce targeted numerical perturbations (a backdoor that activates on specific input patterns), exfiltrate data through covert timing channels, or silently degrade accuracy. You cannot inspect what you cannot read, but you *can* check what it does against what it should do.

These two attack surfaces — weight tampering and kernel compromise — compose badly. If both the weights and the kernel are untrusted, the search space for an auditor explodes. Pinning one of them down with a formal guarantee makes the other tractable.

### 3.11.1 Solution/project Sketch

The weight-loading pipeline should be verified end-to-end: from the filesystem `read()` call, through hash computation and comparison, to the final `cudaMemcpy` (or equivalent) that places weights in device memory. The postcondition is that the bytes in GPU memory are identical to the bytes committed by a known-good hash. This is a refinement proof — show that the runtime representation is a faithful image of the storage representation, and that no intermediate step can silently substitute content. The verified loader then extends the remote attestation chain: the attestation report already covers firmware and boot integrity, so adding a weight-hash measurement means the serving infrastructure can refuse to run inference unless the full chain checks out. Model-checking

the serving state machine (load, attest, serve) confirms that no path reaches the first forward pass without a valid attestation.

For the kernel supply chain, the tool is translation validation [114]. Each kernel gets a behavioral contract: "given input matrices $X$ and $W$, the output is $XW$ up to floating-point error $\varepsilon$." A validator checks the compiled binary against this contract without needing source code — it works on the artifact, which is exactly what you want when the vendor won't ship source. As a stepping stone, differential testing against a formally verified reference implementation (a simple, correct matmul in, say, `Fiat Cryptography`'s [115] style) catches gross violations cheaply. Translation validation catches subtle ones with a proof.

## 3.12 Sampler Integrity and Determinism

*Stack layers:* Software & ML Framework | Execution Harness

*Blocks:* Rogue Insider | Malicious Model

The sampler is the last piece of code between the model's output logits and the token that reaches the user. It applies temperature scaling, top-$k$ or top-$p$ filtering, and then draws from the resulting distribution. This is a small amount of code — a few hundred lines in a typical inference server — but it sits at a chokepoint. An adversary who controls the temperature parameter, the random seed, or the filtering threshold can bias the output distribution without touching weights or the forward pass. The attack surface is not hypothetical: inference servers expose these parameters via API, configuration files, and environment variables, and the sampler's internal state (particularly the PRNG) is rarely isolated from the rest of the serving process.

The scoping here is what makes the problem attractive. Unlike weight loading or kernel validation, the sampler is small enough to verify. The specification is a probability distribution parameterized by (logits, temperature, top-$k$, top-$p$, seed), and the property to check is that the implementation samples from exactly that distribution — no hidden state, no side-channel inputs, no drift from the spec across calls.

### 3.12.1 Solution/project Sketch

Specify the sampler as a probabilistic program and verify it with a probabilistic model checker (`PRISM` [116] or `Storm` [117]). The key property: for any input logit vector, the output token distribution is within a specified KL-divergence bound of the theoretical distribution defined by the declared parameters. This catches both outright bugs (off-by-one in top-$k$ filtering) and subtle manipulation (PRNG state leaking information across requests). Separately, prove data-flow integrity on the implementation — that the sampler is a pure function of its declared inputs with no dependence on mutable global state or memory reachable from other threads. This is a standard information-flow analysis, tractable for code this size. The combination of distributional correctness and data-flow purity gives you a sampler you can trust at the boundary between model and user.

## 3.13 Verified Input Parsers

*Stack layers:* Execution Harness | Software & ML Framework

*Blocks:* External User | Malicious Model

Every boundary in the AI serving stack where text is parsed or transformed is a security boundary. The abuse classifier preprocessor tokenizes, truncates, and encodes user input before classification

— if that pipeline is wrong in any of several subtle ways (non-idempotent truncation, encoding expansion, off-by-one in token windowing), an adversary can craft inputs that the classifier never actually sees. The prompt assembly layer concatenates system instructions, developer instructions, retrieved context, and user messages into a single model input, relying on implicit priority ordering that the model is supposed to respect but cannot enforce. The tokenizer itself — typically byte-pair encoding — is deterministic but has never been formally specified; BPE admits multiple encodings for the same string and includes control-character tokens that downstream components may interpret specially. And on the output side, tool-call parsers extract structured invocations from free-form model text, where any confusion between "model is describing a tool call" and "model is issuing a tool call" is a direct security breach.

These are not four separate problems. They are one problem with four instances: *unverified parsers at trust boundaries*. The LangSec movement [118] has argued for years that ad-hoc input handling is the root cause of most exploitation. In the AI serving stack the argument applies with unusual force, because the inputs are adversarially chosen (user prompts), the transformation pipelines are subtle (BPE, priority-based concatenation), and the consequences of parser confusion range from safety-filter bypass to arbitrary tool execution.

The formal methods tooling for this class of problem is mature. `EverParse` [21] produces verified zero-copy parsers from format specifications, with extraction to C. Parser-combinator libraries in `Lean` and `Coq` can produce proofs of totality, soundness, and completeness alongside executable code. The properties we need — that truncation is idempotent, that encoding is non-expanding, that user-supplied characters cannot occupy system-privileged token positions, that tool-call extraction is sound and complete with respect to a grammar — are all first-order properties over finite structures, well within reach of existing verification technology.

What has been missing is not capability but *targeting*: nobody has written the formal grammars for these specific interfaces and connected them to the actual serving code. The gap is an engineering gap, not a research gap.

### 3.13.1 Solution/project Sketch

Frame this as a family of verification targets sharing a common methodology: for each trust boundary, (1) write a formal grammar of the interface, (2) implement a verified parser (`EverParse` for C-level components, `Lean` parser combinators for higher-level ones), and (3) prove the relevant security properties.

*Abuse classifier interface.* Specify the grammar of valid requests. Verify that the preprocessing pipeline is a total function from the grammar to classifier inputs. Prove truncation is idempotent and encoding is non-expanding — i.e. that `preprocess(preprocess(x)) = preprocess(x)` and `|encode(x)| <= |x|` for the relevant notion of length.

*Prompt assembly.* Define a structured document format with explicit priority zones. Prove non-interference: no user-supplied byte sequence can place tokens into the system-prompt zone. This is a separation property on the assembly function, checkable by symbolic execution or refinement proof.

*Tokenizer.* Formalize BPE in `Lean` or `Coq`. Prove totality (every byte string has exactly one encoding), surjectivity onto the vocabulary, and that detokenization is a left-inverse of tokenization. Define a "safe" token subset and verify that user-controlled input maps only into this subset.

*Tool-call parser.* Define the tool-call grammar as a formal language. Implement as a verified parser. Prove soundness (anything extracted is a valid tool call per the grammar) and completeness (no valid tool call in the output is missed). The parser should reject ambiguous outputs rather than guess.

Each of these is a standalone artifact producing a verified `.c` or `.lean` module that can be linked into existing serving infrastructure. The shared methodology means the second target is cheaper than the first, and the fourth is routine.

## 3.14 Verified Protocol Trust Boundaries

*Stack layers:* Execution Harness | Orchestration & Cloud

*Blocks:* External User | Network MITM

Before any of the AI-specific machinery in this chapter runs, a request has already crossed two or three protocol boundaries that predate AI entirely: the TLS session that carries the HTTPS request, the SSH daemon an operator uses to reach the box, and increasingly the WireGuard overlay that defines which machines can see the inference fleet at all. These are the oldest, most-studied trust boundaries in the stack, and they remain the ones whose single-bug blast radius is largest — Heartbleed [119], FREAK [120], and the Triple Handshake attack [121] on the TLS side; `CVE-2024-6387` (`regreSSHion`) [122], a pre-authentication remote root on the SSH side that turned on a signal-handler race condition, not a parser bug. ARIA's *Safeguarded AI: Cybersecurity* solicitation [76] names verified TLS 1.3, a verified SSH boundary, and a verified WireGuard control plane as three of its six example targets; they belong in this document for the same reason, and they share enough methodology to be one widget rather than three.

The shared shape is the LangSec [118] one: a network-facing boundary parses adversarial bytes and then drives a protocol state machine, and almost every catastrophic bug is either memory unsafety in the parser or an unintended transition in the state machine (downgrade, pre-auth confusion, stale-credential reuse). The formal methods community has already shown each piece is tractable in isolation. `EverParse` [21] produces verified zero-copy binary parsers; Project Everest [123], building on miTLS [124], verified a TLS reference implementation in `F*` down through the HACL*/EverCrypt [125], [126] cryptographic primitives, reaching the TLS 1.3 record layer and a substantial fragment of the handshake; Owl [127] compiles verified secure-channel protocol designs into interoperable implementations; `Tamarin` [107] and `ProVerif` [106] discharge the symbolic protocol-level secrecy and authentication proofs. What none of them deliver is a single machine-checked proof spanning the *whole* boundary — parser, both state machines, and the concurrency and signal-handling discipline of the daemon around them — shipped as a drop-in replacement at a real trust boundary. That gap, not a missing technique, is the widget.

The discipline that keeps this tractable is the assume-the-primitives line. The cryptographic primitives are assumed correct by reusing HACL*/EverCrypt; the OS kernel, hardware, and side channels are out of scope (and partly addressed by Section 3.5 and Section 2.4.2); certificate-path validation is an assumed oracle. What remains is exactly the part the CVE record says actually breaks: wire-format parsing, the RFC state machines, downgrade resistance, and the honest exposure of authentication outcomes to the calling application. This is the same methodology as Section 3.13 — a formal grammar plus a verified parser at a trust boundary — scaled up from in-process text parsing to full network protocols.

### 3.14.1 Solution/project Sketch

Three instances, sharing a parser-plus-state-machine methodology so that the second and third are cheaper than the first.

*Verified TLS 1.3 server.* A protocol-compatible, drop-in TLS 1.3 server with a machine-checked proof covering wire-format parsing, the handshake and record-layer state machines, downgrade resistance, and faithful exposure of authentication outcomes to the application — both server and client roles (the latter for mTLS), a curated cipher-suite set, session resumption, 0-RTT, and post-handshake authentication. The specification: for any sequence of network inputs the implementation either processes input per `RFC 8446` or rejects it; malformed inputs cannot trigger memory unsafety; state machines transition only along `RFC 8446`-permitted paths; negotiated parameters are the strongest mutually supported under policy; and an authenticated session is exposed to the application only when a valid handshake transcript has completed with an authenticated peer. miTLS [124] proved the core; the open frontier is miTLS-grade assurance at production performance and feature-completeness, shippable at the HTTPS edge.

*Verified SSH boundary.* At minimum the packet framing and parsing, the `RFC 4253` transport layer, and the `RFC 4252` user-authentication state machine, including the concurrency and signal-handling discipline of the daemon; a more ambitious version adds channel-opening and multiplexing (`RFC 4254`); the stretch goal is a full protocol-compatible drop-in server. The proof goal is memory safety, state-machine correctness under adversarial input, and the absence of unintended transitions through concurrent or asynchronous control flow — the last is what `regreSSHion` exploited, and what most existing SSH verification work omits. Vest's [128] verified high-performance binary parsers and the HACL*/EverCrypt [125], [126] primitives are the reusable substrate; earlier work formally specifying SSH state machines found that even mature implementations violate parts of the standard [129], but no system covers the full boundary under one proof.

*Verified WireGuard control plane.* The WireGuard data plane (the tunnel itself) is assumed verified — Owl [127] and `Tamarin` [107] have addressed the handshake — so the target is the layer above: the control plane that, given device identities, access policy, and current network state, computes and distributes exactly the peer, key, and routing configuration realising the authorised connectivity graph, across the full device lifecycle (enrolment, approval, key rotation, policy change, revocation). The invariant is a zero-trust containment property: only authenticated and authorised devices can learn about and reach the peers and resources policy permits, preserved under every lifecycle transition. This is the same capability-graph containment analysis as Section 3.7′s agent-authority sketch, applied to network overlay membership rather than gateway request routing. Tailscale and Cloudflare's WARP are the widely deployed but unverified control-plane designs this would displace; the proof target is the control-plane state machine, with the tunnel protocol and identity issuance treated as trusted inputs.

## 3.15 Context Window Integrity Under Adversarial Length

*Stack layers:* Execution Harness

*Blocks:* External User Malicious Model

A model's context window is a fixed-size resource. The serving harness fills it by concatenating, in priority order, a system prompt, developer instructions, retrieved context (from RAG or tool outputs), and the user message. When the user message or retrieved context is long enough,

something has to be truncated — and in many deployed systems, truncation is last-in-wins or proportional, not priority-respecting. An adversary who controls message length (trivially: just send a long message) or who can influence retrieved-context length (less trivially: poison a document store with padding) can force eviction of the system prompt. Once the system prompt is gone, so are the behavioral constraints it encodes.

This is a resource-scheduling problem in disguise. The system prompt and developer instructions are hard reservations; user input and retrieved context are elastic. The invariant is simple to state: *the system prompt occupies a reserved prefix of guaranteed minimum size, and truncation of lower-priority sections never reduces the system-prompt allocation.* This is a monotonicity property — increasing the length of a lower-priority section can only shrink other lower-priority sections, never the reserved prefix. It is exactly the kind of property that tools like `Dafny` and `Frama-C` are built to verify, because it reduces to an invariant over a bounded integer program (token counts, priority levels, allocation sizes).

The reason this has not been done is that context assembly logic is typically embedded in application code — a few dozen lines of Python in a `LangChain` pipeline or a custom harness — and nobody has extracted a specification from it. But the specification is short, the code is short, and the proof obligation is a loop invariant. This is a low-effort, high-value verification target in the serving stack.

### 3.15.1 Solution/project Sketch

Model context allocation as a resource-bounded scheduler with a fixed budget of $N$ tokens and $k$ priority levels. Each level has a minimum reservation $r_i$ with $\sum r_i \leq N$. Surplus tokens are distributed to lower-priority levels in priority order. Specify the allocator in `Dafny` or `Frama-C` and prove two properties: (1) *reservation guarantee* — for all inputs, every priority level receives at least $r_i$ tokens, and (2) *monotonicity* — increasing the requested length of level $j$ does not decrease the allocation of any level $i$ with $i < j$ (higher priority).

Extract the verified allocator as a standalone function that existing harnesses can call to compute truncation boundaries before assembling the context. The function takes token counts per section and returns per-section truncation limits. Integration is a one-line change in any harness that currently does its own ad-hoc truncation.

## 3.16 Capability-Safe Tool Interfaces

*Stack layers:* Execution Harness

*Blocks:* Malicious Model

A single tool call in an agentic harness is usually benign. Read a file, run a shell command, make an HTTP request — each is authorized individually by whatever permission system the harness enforces. The problem is composition: read file + shell exec + HTTP request = exfiltrate a secret key. No single call crossed a policy boundary, but the *sequence* did. This is the weird machines problem [130] applied to AI tool use: individually permitted calls compose into unauthorized capability.

The obvious response — enumerate the tool set, encode it as a process algebra, model-check reachability of a "dangerous state" — has two failure modes that make it a poor shovel-ready target. The dangerous-state predicate is itself the hard part: define it too narrowly (one fixed

exfiltration path) and the verification is trivial; define it too broadly (any bit leaving the session) and every nontrivial tool set admits reachable violations. The analysis also targets the tool set *after* it has been assembled — by the time the model checker runs, composition has already happened, and the verifier's only lever is to refuse to ship. A "your agent is weird-machine-complete" diagnosis arrives too late to inform the design.

The alternative is to stop asking "is this tool set safe" and start asking "what does a tool interface look like such that safe composition is the default". Object-capability languages — `E`, `Newspeak`, `Monte`, and at the OS level `Capsicum` [131] and `seL4`'s capability system [84] — have decades of theory on this question, and none of it has been specialized to AI tool dispatch. ARIA's *Safeguarded AI: Cybersecurity* solicitation [76] names a "verified capability-mediated runtime for AI agents" — each agent in its own compartment holding only explicit, unforgeable capability tokens, with `seL4`, `capDL`, and `Microkit` as the cited substrate — as an example target, converging on the same object-capability framing from the OS-runtime side that this widget approaches from the tool-interface side. The ambient-authority assumption pervading current tool protocols (a tool named `file_read` taking a path argument implicitly holds read authority over the entire filesystem, attenuated only by whatever the harness decides to do post-hoc) is exactly the assumption those languages refuse.

This widget is the correct-by-construction arm of the two-arm split von Hippel draws [132] between correct-by-construction tool languages and cheap-proofs-per-deployment. It is worth pursuing not because it displaces the per-deployment path, but because writing down what "safe composition" means in an interface forces the vocabulary that the per-deployment path also depends on. See Section 3.19 for the enabler that carries that vocabulary.

### 3.16.1 Solution/project Sketch

The deliverable is a tool-interface specification language with two properties. Every tool declares its capability footprint as a typed requirement — what authority it needs, expressed as a labelled region of the session's capability set, not an ambient permission. Composition of tools is constrained by a flow-type system such that any well-typed agent program satisfies a declared non-exfiltration theorem by construction.

Three pieces.

First, a surface language. Extend an existing tool-declaration format (`MCP` schemas, `OpenAI` function-calling JSON) with capability annotations: each parameter carries a flow label, each return value carries a label, and the tool's behavior is a declared transformation between them. This is `Jif`-style [133] information-flow typing retrofitted onto tool APIs, with the additional constraint that authority is held as first-class capabilities (read-this-file, post-to-this-URL) rather than pool permissions (read-any-file).

Second, a type system. Given a set of annotated tools and an agent program (a trace or policy that composes them), the type checker accepts only compositions in which no high-label value flows to a low-label sink. The novelty is in the labels: they are capability references, not secrecy classes, so the non-interference theorem states that an attacker-reachable channel can only carry data whose provenance the capability graph permits.

Third, a reference implementation and soundness proof. Build the type checker in `Rocq` or `Lean`, prove progress-and-preservation against a small-step semantics of the tool-dispatch loop, and ship

a compiler from the surface language to a thin runtime wrapper over an existing agent harness. A well-typed program cannot, by construction, compose the file-read + shell-exec + HTTP-post exfiltration chain, because the flow labels on the file contents do not match the labels the HTTP sink accepts — not because the harness noticed and blocked the call, but because the program would not have type-checked.

What the project does *not* do: it does not decide which capabilities to grant an agent for a given user task. That is a policy question, owned by whoever writes the agent's capability manifest. What the project delivers is a tool interface in which the manifest has teeth — stated capabilities bound behavior by construction, not by runtime check.

## 3.17 Audit Log Integrity and Session State Channels

*Stack layers:* Execution Harness Orchestration & Cloud

*Blocks:* Malicious Model Rogue Insider

The audit log is the primary accountability mechanism for deployed AI systems. Regulators, red teams, and incident responders all depend on the same thing: a complete, tamper-evident record of what the model did. If log entries can be suppressed, falsified, or selectively deleted, the system is unauditable — and any safety claim that rests on "we can review the logs" collapses. Append-only storage and cryptographic log chaining (in the style of Certificate Transparency [134]) provide tamper *evidence*, but the implementations are not formally verified. The gap between "we use Merkle trees" and "no API call sequence available to the serving infrastructure can modify a committed entry" is exactly the gap formal methods fills.

There is a subtler problem on the same boundary. When a model's output or summary is written back to session state for future context windows, a persistent side channel is created. A misaligned model can encode messages in session state that survive request boundaries — instructions to future invocations of itself, breadcrumbs for capability accumulation, or data staged for later exfiltration. This channel is invisible to the output filter, which inspects the response returned to the user, not the write-back to the session store. The threat model is not hypothetical: any system that persists conversation summaries, scratchpads, or tool-call results across requests has this surface.

The two problems are linked by a common structure: both involve a write path that must satisfy integrity constraints, and both fail silently when the constraints are violated. A corrupted log and a poisoned session store both look normal from outside the system until someone audits the contents — and the audit itself depends on the log being intact.

### 3.17.1 Solution Sketch A: Log Integrity via Permission Graph

Build a `TLA+` model of the serving infrastructure's request lifecycle. State variables: the committed log, the in-flight request set, the per-principal permission relation. Transitions: a request enters the gateway, dispatches to forward-pass execution, emits a log entry, commits the entry to storage, returns a response. Two properties discharge against the same model. *Tampering safety*: in every reachable state, no transition available to any principal (model, harness, orchestrator) modifies or deletes an already-committed log entry — a safety check over the permission relation and the storage state machine. *Logging completeness*, the liveness counterpart: every request that reaches forward-pass execution is eventually followed by a log-commit transition before the corresponding

response-emit transition, catching suppression by the same infrastructure that handles writing. Sketch A treats the log as a passive resource and verifies the surrounding access-control and lifecycle machinery that protects it.

### 3.17.2 Solution Sketch B: Log Integrity via Verified Merkle-Tree Log

An alternative architecture closes the same gap from the other direction: rather than verifying that the permission system prevents tampering, make tampering cryptographically detectable inside the log itself. Append-only Merkle-tree logs of the kind Certificate Transparency [134] uses (also `sigstore`'s `Rekor` [28]) publish signed tree heads that any third party can use to check, via consistency proofs, that two published heads belong to the same history. A serving infrastructure that retroactively removes a committed entry cannot then issue a new signed head consistent with the previous one — the tamper-evidence rests on the structure, not on the access-control system.

The verification target is the log component itself, against an abstract specification of an append-only Merkle log. Three local properties:

- The `append(entry)` operation produces a tree whose root is a valid extension of the previous root; no implementation path produces a signed root that omits a previously committed entry.
- `consistency_proof(old_root, new_root)` is sound (it only verifies for genuinely-consistent trees) and complete (it verifies whenever the trees are consistent).
- The signing operation has no path that produces a signed tree head for a tree the log did not append to in order.

These are local properties of a single program and discharge cleanly in `Dafny` or `F*` as a refinement against the abstract spec. The result is a verified log component whose tamper-evidence guarantees rest on a small cryptographic structure rather than the permission graph surrounding it — a property that any third party with two signed tree heads can re-check independently of the operator, which Sketch A's approach cannot offer. Sketch A and Sketch B are alternatives: pick one (or run both, treating Sketch B's verified log as the storage primitive Sketch A's permission model protects).

### 3.17.3 Solution Sketch C: Session State via Noninterference

For the persistent-side-channel problem, apply information flow analysis. Model the session store as shared memory and the write-back function as the interface under test. The noninterference property: the content written to session state must be a function only of authorized inputs (the model's declared output schema, the user's message, the tool results). Any information flow from the model's hidden state to the session store that is not mediated by the declared output path is a violation. This is verifiable as a pre/postcondition contract on the write-back function — specify the allowed content, prove the function produces only that content. `Dafny` is a natural fit: the write-back function is small, the specification is a refinement type on the output, and the verifier handles the quantifier reasoning automatically.

## 3.18 Governance and Interpretability of Neuralese Proof Stacks

*Stack layers:* Execution Harness | Software & ML Framework | Orchestration & Cloud

*Blocks:* Malicious Model

table stakes is that proofs are cheap. We assume a proof oracle will make code and proofs that they're correct too cheap to meter.

One idea is that specs remain expensive, though, because we always want a human in the loop. Elicitation and validation of specs is really crucial.

But in the "specs are expensive, proofs are cheap" world, we assume some known trusted natural/human-written proof stack, like `Lean` or `Rocq` or `Verus/Z3`, is the substrate. What if this assumption is violated?

What if LeoGPT writes a new proof stack, where human interpretability is not a first class feature?

how do we do spec elicitation and validation *then*? how do we govern the proof oracle if we've never seen the syntax before?

TODO: flesh out.

## 3.19 Formal Theory of Weird Machines in Agents

*Stack layers:* Execution Harness Orchestration & Cloud

*Blocks:* Malicious Model

A weird machine [130] is the unintended computational substrate an attacker programs when individually-legitimate operations compose into capabilities their designers never enumerated. LangSec [118] named this and argued it is the root cause of most exploitation. Von Hippel [132] observes that the framing transfers directly to agentic harnesses: the "gadgets" are the tools the agent is permitted to call (`curl`, `cat`, `man`, a Python sandbox, a vector-store query), and the weird machine is whatever computation the tool set admits under composition. Adversarial agents will live off the land, and "is this permission set safe?" reduces to a question about the reachable states of a composed transition system no one has ever formally described.

Section 3.16 takes this seriously as a widget: specify a capability-safe tool interface in which well-typed compositions cannot reach exfiltration states by construction. That widget ships a type system and a soundness theorem; what it does not ship is the capability vocabulary the type system needs — which labels matter, what counts as "reachable" across agent boundaries, what threshold separates safe composition from unsafe. That is the theoretical gap this enabler addresses.

The missing theory has two pieces. First, a notion of *capability threshold*: at what point does adding a tool push the composed system past a line that matters? Turing-completeness is the obvious threshold but not quite the right one — a tool set that is Turing-complete over session-local state can still be safe if it cannot reach exfiltration channels, and a sub-Turing set can be lethal if one primitive is a `POST` to an attacker-controlled endpoint. We need a capability-theoretic analogue of complexity classes, indexed by what the tool set can *observably do to the world* rather than what it can compute internally. Second, an inter-agent extension: once agents can message one another, the composition problem is distributed and gadgets cross agent boundaries. The classical weird-machines literature has no distributed analogue.

Von Hippel sketches two paths and picks one. The path he rejects (fn. 8 of [132]) is classical-LangSec-for-agents: cast tool invocations into a correct-by-construction language with inherent safety or runtime-observability guarantees. Meijer's *Guardians of the Agents* [135] is the cleanest current instantiation of that path: it argues that prompt injection is a code-data confusion problem of the same shape as SQL injection, has the LLM emit a structured plan over symbolic references rather than calling tools one at a time, and discharges the plan against a security policy with

three independent static checks (taint analysis, security automata, and `Z3`) before any side-effecting tool runs. The path von Hippel prefers leans on cheap proofs to discharge safety obligations per deployment. The second is probably the right long-run bet, but it inherits its soundness from the theory being articulated here: proof oracles need to know what they are proving, and "this permission set does not induce a dangerous weird machine" is not currently a statement anyone can write down rigorously. The correct-by-construction path is still worth understanding, if only because it forces you to name the safety properties the proof oracles will be asked to verify.

This is an enabler rather than a widget because it does not ship a verified artifact. It produces a vocabulary — capability classes, threshold lemmas, decidability results for bounded fragments — that downstream widgets (Section 3.16, Section 3.13) can target. Until that vocabulary exists, those widgets are verifying properties whose significance we can only argue informally.

### 3.19.1 Solution/project Sketch

Three threads, in rough dependency order.

*Capability-class hierarchy.* Adapt the existing process-algebra and IFC literature into a taxonomy of tool-set capabilities indexed by observable effects (read, write, exfiltrate, persist, escalate, signal). Formalize "tool set $T$ induces capability $C$" as a reachability predicate on the composed transition system. Identify fragments where the predicate is decidable (finite data, bounded steps, no unbounded recursion) and fragments where it is not. Deliverable: a `Lean` or `Rocq` library of definitions and a handful of threshold theorems.

*Distributed extension.* Lift the above to multi-agent settings where agents share a message channel. The relevant analogues are concurrent separation logic and session types — the question becomes whether any protocol reachable from the permitted message grammar induces a capability the individual agents lack. Expect most interesting cases to be undecidable; the deliverable is a soundly over-approximating type system.

*Undecidability boundaries.* Prove Rice-style results pinning down where automated analysis of tool-set safety is and is not possible. This matters operationally: it tells harness designers which guarantees they can buy with static checking and which require runtime monitoring or per-session proof-oracle discharge.

The output of all three is papers and libraries, not deployable C code. That is the point of an enabler.

# 4 Appendix: Adversary taxonomy

The *Invites* tags on each stack layer (Section 2) and the *Blocks* tags on each tractable problem (Section 3) draw from a fixed set of adversary archetypes. The same taxonomy populates the tooltips on the website. Each archetype is named for the position it occupies relative to the target system, not for a specific actor or organization.

| Adversary | Description |
|---|---|

| | |
|---|---|
| Supply Chain | Malicious or compromised upstream: poisoned packages (PyPI/npm), tampered model weights, trojaned chips, EDA-tool insertions, counterfeit firmware. |
| Co-tenant | A workload sharing infrastructure with the victim — same GPU partition, same K8s node, same NVLink fabric — that exploits weak isolation to read or corrupt the victim's state. |
| Malicious Model | The model or agent itself acting adversarially, whether misaligned, jailbroken, or scheming: escapes its sandbox, composes tool calls into capabilities it was not granted, or emits inputs that exploit downstream parsers. |
| Rogue Insider | A principal with legitimate access abusing it: stolen signing keys, over-permissioned service accounts, push access to a model registry, BMC takeover. |
| Network MITM | An on-path attacker between nodes or between client and server: tampering unauthenticated RDMA traffic, compromising an SDN controller, or performing traffic analysis on collective-ops patterns. |
| Physical Adversary | An attacker with physical access to hardware: probing HBM interposers, EM and power side channels, counterfeit or remarked chips, tampering with server chassis. |
| External User | An external actor hitting the public API: crafted prompts that trigger deserialization bugs, content-filter evasion, or state-poisoning of rate-limiters and context windows. |

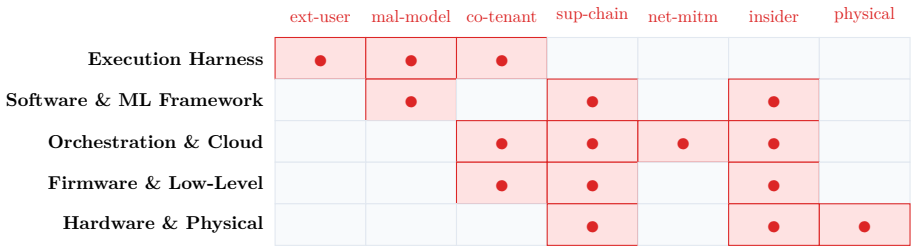


Figure 9: Which adversary archetypes each stack layer invites. The same `// Invites:` headers drive the per-layer tags in Section 2; this matrix is just an at-a-glance view of the bipartite relation.

# Bibliography

[1] J. Regehr, "Zero-Degree-of-Freedom LLM Coding using Executable Oracles." [Online]. Available: https://john.regehr.org/writing/zero_dof_programming.html

[2] M. Kleppmann, "Prediction: AI will make formal verification go mainstream." [Online]. Available: https://martin.kleppmann.com/2025/12/08/ai-formal-verification.html

[3] M. von Hippel, "Secure Program Synthesis." [Online]. Available: https://www.lesswrong.com/posts/8wtrLoDPyCfMLuHkt/how-to-solve-secure-program-synthesis

[4] G. P. Sarma, R. Steratore, S. D. Bhatt, and G. Irving, "Verified Machine Learning Infrastructure: Formal Methods for Trustworthy Artificial Intelligence Deployment," Santa Monica, CA, Research Report RR-A4881-1, June 2026. doi: 10.7249/RRA4881-1.

[5] The White House, "Promoting Advanced Artificial Intelligence Innovation and Security." [Online]. Available: https://www.whitehouse.gov/presidential-actions/2026/06/promoting-advanced-artificial-intelligence-innovation-and-security/

[6] Anthropic, "Anthropic's Frontier Safety Roadmap." [Online]. Available: https://www.anthropic.com/responsible-scaling-policy/roadmap

[7] BlueRock Security, "NOVA: A Microhypervisor-Based Secure Virtualization Architecture." [Online]. Available: https://bluerocksec.gitlab.io/formal-methods/faq/what-is-nova/

[8] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal Methods: Practice and Experience," *ACM Computing Surveys*, vol. 41, no. 4, pp. 1–36, 2009, doi: 10.1145/1592434.1592436.

[9] ggml-org, "GHSA-96jg-mvhq-q7q7: Heap Buffer Overflow via Integer Overflow in GGUF Tensor Parsing." [Online]. Available: https://github.com/ggml-org/llama.cpp/security/advisories/GHSA-96jg-mvhq-q7q7

[10] Trail of Bits, "EleutherAI, Hugging Face Safetensors Library Security Assessment," technical report, May 2023. [Online]. Available: https://github.com/trailofbits/publications/blob/master/reviews/2023-03-eleutherai-huggingface-safetensors-securityreview.pdf

[11] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984, doi: 10.1145/358198.358210.

[12] LLVM Project, "CVE-2023-29932 through CVE-2023-29939: Memory-management vulnerabilities in LLVM MLIR." 2023.

[13] Anthropic, "A postmortem of three recent issues." [Online]. Available: https://www.anthropic.com/engineering/a-postmortem-of-three-recent-issues

[14] S. Bhat, A. d. Keizer, C. Hughes, A. Goens, and T. Grosser, "Verifying Peephole Rewriting in SSA Compiler IRs," in *15th International Conference on Interactive Theorem Proving (ITP 2024)*, 2024. doi: 10.4230/LIPIcs.ITP.2024.9.

[15] M. Fehr, Y. Fan, H. Pompougnac, J. Regehr, and T. Grosser, "First-Class Verification Dialects for MLIR," in *Proceedings of the ACM on Programming Languages (PLDI)*, 2025. doi: 10.1145/3729309.

[16] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009, doi: 10.1145/1538788.1538814.

[17] PyTorch Contributors, "CVE-2025-32434: Remote code execution via \texttt{torch.load} with \texttt{weights_only=True}." 2025.

[18] JFrog Security Research, "Unveiling 3 Zero-Day Vulnerabilities in PickleScan." [Online]. Available: https://jfrog.com/blog/unveiling-3-zero-day-vulnerabilities-in-picklescan/

[19] ReversingLabs, "Malicious ML models discovered on Hugging Face platform." [Online]. Available: https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face

[20] Google / TensorFlow, "CVE-2024-3660: Arbitrary code execution via Keras Lambda layer \texttt{safe_mode} bypass." 2024.

[21] T. Ramananandro *et al.*, "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[22] K. Zhang, Y. Feng, and Z. Chen, "Unraveling the Characterization and Propagation of Security Vulnerabilities in TensorFlow-based Deep Learning Software Supply Chain," in *16th Asia-Pacific Symposium on Internetware (Internetware 2025)*, 2025. doi: 10.1145/3755881.3755923.

[23] J. Mahon, C. Hou, and Z. Yao, "PyPitfall: Dependency Chaos and Software Supply Chain Vulnerabilities in Python," 2025.

[24] PyTorch Team, "Compromised PyTorch-nightly dependency chain (torchtriton)." [Online]. Available: https://pytorch.org/blog/compromised-nightly-dependency/

[25] ReversingLabs, "Compromised Ultralytics PyPI package delivers crypto coin-miner." [Online]. Available: https://www.reversinglabs.com/blog/compromised-ultralytics-pypi-package-delivers-crypto-coinminer

[26] FutureSearch, "litellm Supply Chain Attack." [Online]. Available: https://futuresearch.ai/blog/litellm-pypi-supply-chain-attack/

[27] SLSA Contributors, "SLSA v1.0: Supply-chain Levels for Software Artifacts." [Online]. Available: https://slsa.dev/spec/v1.0/

[28] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: Software Signing for Everybody," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022. doi: 10.1145/3548606.3560596.

[29] A. Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies." [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[30] NVIDIA, "CVE-2024-0132: NVIDIA Container Toolkit TOCTOU container escape." 2024.

[31] Open Containers / Snyk Security Labs, "CVE-2024-21626: runc container escape via leaked file descriptors (Leaky Vessels)." 2024.

[32] SchedMD, "CVE-2023-49935: Slurm MUNGE message integrity bypass." 2023.

[33] SchedMD, "CVE-2022-29501: Slurm PMI2/PMIx handler allows unprivileged write to arbitrary Unix sockets." 2022.

[34] SchedMD, "CVE-2017-15566: Slurm SPANK environment variable privilege escalation." 2017.

[35] Oligo Security, "CVE-2023-48022: Unauthenticated remote code execution in Ray (ShadowRay)." 2023.

[36] The Kubernetes Authors, "Schedule GPUs." [Online]. Available: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

[37] C. J. Anderson *et al.*, "NetKAT: Semantic Foundations for Networks," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2014, pp. 113–126. doi: 10.1145/2535838.2535862.

[38] K. Benton, L. J. Camp, and C. Small, "OpenFlow Vulnerability Assessment," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2013, pp. 151–152. doi: 10.1145/2491185.2491222.

[39] NVIDIA, "NVIDIA Hopper Architecture In-Depth." [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[40] W. Li *et al.*, "Analyzing Communication Predictability in LLM Training," 2025.

[41] NVIDIA, "NCCL User Guide: Setup." [Online]. Available: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/setup.html

[42] Sysdig Threat Research Team, "AI-assisted cloud intrusion achieves admin access in 8 minutes." [Online]. Available: https://sysdig.com/blog/ai-assisted-cloud-intrusion-achieves-admin-access-in-8-minutes

[43] Mithril Security, "PoisonGPT: How we hid a lobotomized LLM on Hugging Face to spread fake news." [Online]. Available: https://blog.mithrilsecurity.io/poisongpt-how-we-hid-a-lobotomized-llm-on-hugging-face-to-spread-fake-news/

[44] GitGuardian, "The State of Secrets Sprawl 2025," technical report, 2025. [Online]. Available: https://www.gitguardian.com/state-of-secrets-sprawl-report-2025

[45] S. Biggs, D. Lee, and G. Heiser, "The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security," in *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*, 2018. doi: 10.1145/3265723.3265733.

[46] F. Wilhelm, "An EPYC escape: Case-study of a KVM breakout." [Online]. Available: https://projectzero.google/2021/06/an-epyc-escape-case-study-of-kvm.html

[47] Trail of Bits, "CVE-2023-4969: LeftoverLocals — GPU local memory leak across tenant boundaries." 2023.

[48] Y. Zhang, R. Nazaraliyev, S. B. Dutta, A. Marquez, K. Barker, and N. Abu-Ghazaleh, "NVBleed: Covert and Side-Channel Attacks on NVIDIA Multi-GPU Interconnect." [Online]. Available: https://arxiv.org/abs/2503.17847

[49] Advanced Micro Devices, "AMD SEV-TIO: Trusted I/O for Secure Encrypted Virtualization." [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/developer/sev-tio-whitepaper.pdf

[50] NVIDIA, "CVE-2024-0126: NVIDIA GPU Display Driver privilege escalation." 2024.

[51] NVIDIA, "CVE-2024-0107: NVIDIA GPU Display Driver out-of-bounds read." 2024.

[52] A. Zambelli, "Multiple Vulnerabilities Discovered in NVIDIA CUDA Toolkit." [Online]. Available: https://unit42.paloaltonetworks.com/nvidia-cuda-toolkit-vulnerabilities/

[53] NVIDIA, "CVE-2022-21819: NVIDIA Jetson PCIe DMA attack bypassing secure boot." 2022.

[54] NVIDIA, "NVIDIA Releases Open-Source GPU Kernel Modules." [Online]. Available: https://developer.nvidia.com/blog/nvidia-releases-open-source-gpu-kernel-modules/

[55] G. Stewart, "Verified ZynqMP DMA Driver in Concurrent Separation Logic (talk)." [Online]. Available: https://sel4.systems/Summit/2025/abstracts2025.html#a-verified-zynqmp

[56] J. Haag, Y. Hirai, S. Hudon, A. Masood, G. Malecha, and G. Stewart, "Protocol Completion of a Robust C++ Virtual Switch," Tech report, Aug. 2024. Accessed: June 14, 2026. [Online]. Available: https://bluerocksec.gitlab.io/formal-methods/tech_reports/protocol-completion-of-a-robust-c-virtual-switch/

[57] BlueRock Security, "Verifying a Virtual Machine Monitor," Tech report, 2024. [Online]. Available: https://bluerocksec.gitlab.io/formal-methods/tech_reports/verifying-a-virtual-machine-monitor/

[58] BlueRock Security, "Modularizing CPU Semantics for Virtualization," Tech report, 2024. [Online]. Available: https://bluerocksec.gitlab.io/formal-methods/tech_reports/modularizing-cpu-semantics-for-virtualization/

[59] P. Sewell and REMS Group, "REMS: Rigorous Engineering of Mainstream Systems." [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/

[60] B. Simner, A. Armstrong, J. Pichon-Pharabod, C. Pulte, R. Grisenthwaite, and P. Sewell, "Relaxed Virtual Memory in Armv8-A," in *Programming Languages and Systems (ESOP 2022)*, 2022. doi: 10.1007/978-3-030-99336-8_6.

[61] Microsoft, "CVE-2022-21894: UEFI Secure Boot bypass exploited by BlackLotus bootkit." 2022.

[62] Binarly, "CVE-2023-40284 through CVE-2023-40290: Supermicro BMC IPMI firmware vulnerabilities." 2023.

[63] BleepingComputer, "NVIDIA confirms data was stolen in recent cyberattack." [Online]. Available: https://www.bleepingcomputer.com/news/security/nvidia-confirms-data-was-stolen-in-recent-cyberattack/

[64] A. Classen, D. Hansma, T.-N. Ngo, M. Becker, and K. Rieck, "Evil from Within: Machine Learning Backdoors through Hardware Trojans," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2025. [Online]. Available: https://arxiv.org/abs/2304.08411

[65] K. Basu *et al.*, "CAD-Base: An Attack Vector into the Electronics Supply Chain," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 4, pp. 38:1–38:30, 2019, doi: 10.1145/3315574.

[66] J. Harrison, "Formal Verification at Intel," in *18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2003, pp. 45–54. doi: 10.1109/LICS.2003.1210044.

[67] A. Reid, "Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD)*, 2016. doi: 10.1109/FMCAD.2016.7886675.

[68] YosysHQ, "riscv-formal: RISC-V Formal Verification Framework." [Online]. Available: https://github.com/YosysHQ/riscv-formal

[69] OpenHW Group, "CORE-V-VERIF: Functional Verification for CORE-V RISC-V Cores." [Online]. Available: https://github.com/openhwgroup/core-v-verif

[70] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification," *Proceedings of the ACM on Programming Languages (ICFP)*, vol. 1, 2017, doi: 10.1145/3110268.

[71] NVIDIA, "Confidential Compute on NVIDIA Hopper H100." [Online]. Available: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/HCC-Whitepaper-v1.0.pdf

[72] H. Maia, C. Chang, D. Sanchez, and S. Devadas, "Can one hear the shape of a neural network?: Snooping the GPU via magnetic side channel," in *Proceedings of the 31st USENIX Security Symposium*, 2022.

[73] Z. Zhan *et al.*, "Graphics Peeping Unit: Exploiting EM Side-Channel Information of GPUs," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022. doi: 10.1109/SP46214.2022.9833773.

[74] Semiconductor Industry Association, "Winning the Battle Against Counterfeit Semiconductor Products." [Online]. Available: https://www.semiconductors.org/wp-content/uploads/2018/01/SIA-Anti-Counterfeiting-Whitepaper.pdf

[75] Semiconductor Industry Association and Boston Consulting Group, "Strengthening the Global Semiconductor Supply Chain in an Uncertain Era," technical report, Apr. 2021. [Online]. Available: https://www.semiconductors.org/strengthening-the-global-semiconductor-supply-chain-in-an-uncertain-era/

[76] Advanced Research and Invention Agency, "Safeguarded AI: Cybersecurity — Call for Proposals." [Online]. Available: https://aria.org.uk/media/1padxpaf/safeguarded-ai_cybersecurity_solicitation.pdf

[77] L. de Moura, "Who Watches the Provers?." [Online]. Available: https://leodemoura.github.io/blog/2026-3-16-who-watches-the-provers/

[78] J. Harrison, "HOL Light." [Online]. Available: https://hol-light.github.io/

[79] M. Adams, "HOL Zero." [Online]. Available: http://www.proof-technologies.com/holzero/

[80] CakeML Project, "Candle: A Self-Compiling Verification of HOL Light." [Online]. Available: https://cakeml.org/candle/

[81] J. Davis, "The Milawa Rewriter and an ACL2 Proof of its Soundness," in *Proceedings of the Seventh International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '07)*, Austin, Texas, USA, Oct. 2007. [Online]. Available: https://kookamara.com/jared/milawa/Documentation/Rewriter/rewrite.pdf

[82] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2000, pp. 268–279. doi: 10.1145/351240.351266.

[83] G. Ammons, R. Bodík, and J. R. Larus, "Mining Specifications," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002, pp. 4–16. doi: 10.1145/503272.503275.

[84] G. Klein *et al.*, "seL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 207–220.

[85] K. Fisher, J. Launchbury, and R. Richards, "The HACMS Program: Using Formal Methods to Eliminate Exploitable Bugs," *Philosophical Transactions of the Royal Society A*, vol. 375, no. 2104, p. 20150401, Oct. 2017, doi: 10.1098/rsta.2015.0401.

[86] A. Lyons, K. McLeod, H. Almatary, and G. Heiser, "Scheduling-Context Capabilities: A Principled, Light-Weight Operating-System Mechanism for Managing Time," in *Proceedings of the Thirteenth EuroSys Conference (EuroSys 2018)*, 2018, pp. 1–16. doi: 10.1145/3190508.3190539.

[87] BlueRock Formal Methods, "NOVA is Getting Relaxed-Memory Ready (26.08.0)." Accessed: June 14, 2026. [Online]. Available: https://bluerocksec.gitlab.io/formal-methods/blogs/2026-03-31-NOVA-relaxed-memory/

[88] A. Saidi, "AWS Nitro Isolation Engine: Formally Verifying the Hypervisor in the AWS Nitro System." Accessed: June 14, 2026. [Online]. Available: https://aws.amazon.com/blogs/compute/aws-nitro-isolation-engine-formally-verifying-the-hypervisor-in-the-aws-nitro-system/

[89] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A Secure and Formally Verified Linux KVM Hypervisor," in *2021 IEEE Symposium on Security and Privacy (S&P)*, May 2021, pp. 1782–1799. doi: 10.1109/SP40001.2021.00049.

[90] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems," *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, 2007, doi: 10.1016/j.jss.2006.08.039.

[91] Kry10 Unlimited, Inc., "Kry10: A Secure OS for Software Defined Machines." Accessed: June 14, 2026. [Online]. Available: https://kry10.com/

[92] G. Malecha, H.-H. Dang, P. G. Giarrusso, S. Hudon, J.-O. Kaiser, and D. Swasey, "Modular, Full-System Verification," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS 2025)*, 2025, pp. 42–49. doi: 10.1145/3713082.3730387.

[93] R. Marchand *et al.*, "Quantifying Frontier LLM Capabilities for Container Sandbox Escape," *arXiv preprint arXiv:2603.02277*, 2026.

[94] UK AI Safety Institute, "AISI Sandboxing Toolkit." 2025.

[95] UK AI Safety Institute, "Docker Sandbox Breakout." 2026.

[96] UK AI Safety Institute, "Inspect AI." 2025.

[97] A. Kaufman, J. Lucassen, T. Tracy, C. Rushing, and A. Bhatt, "BashArena: A Control Setting for Highly Privileged AI Agents," *arXiv preprint arXiv:2512.15688*, 2025.

[98] Q. Dougherty, A. Singh, and M. von Hippel, "BoxArena: An OCI Runtime Escape Benchmark." 2026.

[99] W. R. Bevier, W. A. Hunt, J. S. Moore, and W. D. Young, "An Approach to Systems Verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 411–428, 1989.

[100] R. Greenblatt, B. Shlegeris, K. Sachan, and F. Roger, "AI Control: Improving Safety Despite Intentional Subversion," in *International Conference on Machine Learning (ICML)*, 2024.

[101] A. Bhatt *et al.*, "Ctrl-Z: Controlling AI Agents via Resampling," *arXiv preprint arXiv:2504.10374*, 2025.

[102] G. C. Necula, "Proof-Carrying Code," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1997, pp. 106–119. doi: 10.1145/263699.263712.

[103] Q. Dougherty, "Formal Confinement." 2025.

[104] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983, doi: 10.1109/TIT.1983.1056650.

[105] Auth0, "CVE-2015-9235: JWT algorithm confusion in \texttt{jsonwebtoken}." 2015.

[106] B. Blanchet, "An Efficient Cryptographic Protocol Verifier Based on Prolog Rules," in *14th IEEE Computer Security Foundations Workshop (CSFW)*, 2001, pp. 82–96.

[107] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The TAMARIN Prover for the Symbolic Analysis of Security Protocols," in *25th International Conference on Computer Aided Verification (CAV)*, in Lecture Notes in Computer Science, vol. 8044. 2013, pp. 696–701. doi: 10.1007/978-3-642-39799-8_48.

[108] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[109] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015. doi: 10.1109/SP.2015.43.

[110] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, pp. 256–290, 2002, doi: 10.1145/505145.505149.

[111] M. Moeller *et al.*, "KATch: A Fast Symbolic Verifier for NetKAT," *Proceedings of the ACM on Programming Languages (PLDI)*, vol. 8, 2024, doi: 10.1145/3656454.

[112] N. Cankaya, A. Friedman, and M. Baker, "Research Note: The Fundamentals and Feasibility of Secure Network Taps for Verifying AI Datacenter Use." [Online]. Available: https://nacicankaya.substack.com/p/research-note-the-fundamentals-and

[113] Amodo Design, "Network Tapping for AI Verification: A Technical Assessment." [Online]. Available: https://amododesign.com/network-tapping-tech-note/

[114] A. Pnueli, M. Siegel, and E. Singerman, "Translation Validation," in *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1998, pp. 151–166. doi: 10.1007/BFb0054170.

[115] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple High-Level Code for Cryptographic Arithmetic – With Proofs, Without Compromises," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019. doi: 10.1109/SP.2019.00005.

[116] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of Probabilistic Real-Time Systems," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011, pp. 585–591. doi: 10.1007/978-3-642-22110-1_47.

[117] C. Dehnert, S. Junges, J.-P. Katoen, and M. Volk, "A Storm is Coming: A Modern Probabilistic Model Checker," *International Journal on Software Tools for Technology Transfer (STTT)*, 2017, doi: 10.1007/s10009-017-0469-y.

[118] L. Sassaman, M. L. Patterson, S. Bratus, M. E. Locasto, and A. Shubina, "Security Applications of Formal Language Theory," 2013, pp. 489–500.

[119] Z. Durumeric *et al.*, "The Matter of Heartbleed," in *Proceedings of the 2014 Internet Measurement Conference (IMC)*, 2014, pp. 475–488. doi: 10.1145/2663716.2663755.

[120] B. Beurdouche *et al.*, "A Messy State of the Union: Taming the Composite State Machines of TLS," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 535–552. doi: 10.1109/SP.2015.39.

[121] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub, "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 98–113. doi: 10.1109/SP.2014.14.

[122] Qualys Threat Research Unit, "regreSSHion: Remote Unauthenticated Code Execution Vulnerability in OpenSSH Server (CVE-2024-6387)." [Online]. Available: https://blog.qualys.com/vulnerabilities-threat-research/2024/07/01/regresshion-remote-unauthenticated-code-execution-vulnerability-in-openssh-server

[123] K. Bhargavan *et al.*, "Everest: Towards a Verified, Drop-in Replacement of HTTPS," in *2nd Summit on Advances in Programming Languages (SNAPL)*, 2017. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.SNAPL.2017.1

[124] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with Verified Cryptographic Security," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[125] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017. [Online]. Available: https://eprint.iacr.org/2017/536

[126] J. Protzenko *et al.*, "EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider," in *IEEE Symposium on Security and Privacy (S&P)*, 2020. [Online]. Available: https://eprint.iacr.org/2019/757

[127] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "Owl: Compositional Verification of Security Protocols via an Information-Flow Type System," in *IEEE Symposium on Security and Privacy (S&P)*, 2023. [Online]. Available: https://eprint.iacr.org/2023/473

[128] Y. Cai *et al.*, "Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust," in *USENIX Security Symposium*, 2025. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/cai-yi

[129] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model Learning and Model Checking of SSH Implementations," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, 2017, pp. 142–151. doi: 10.1145/3092282.3092289.

[130] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation," 2011.

[131] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[132] M. von Hippel, "Scalable Formal Oversight: Open Problems in AI Safety and Formal Methods." [Online]. Available: https://www.lesswrong.com/posts/SfhFh9Hfm6JYvzbby/the-scalable-formal-oversight-research-program

[133] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," 1999.

[134] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," *RFC 6962*, 2013.

[135] E. Meijer, "Guardians of the Agents: Formal Verification of AI Workflows," *Communications of the ACM*, vol. 69, no. 1, pp. 46–52, Jan. 2026, doi: 10.1145/3777544.