Don't Trust the Code, Check Its Effects

Runtime Refinement for Regenerated Systems Code Under an Adversarial Generator

Jinhao Hu
Max Planck Institute for Software Systems
Saarbrücken, Germany

Ashvin Goel
University of Toronto
Toronto, Canada

Laurent Bindschaedler
Max Planck Institute for Software Systems
Saarbrücken, Germany

Abstract

Recent work uses large language models to generate systems code from specifications, treating the specification as the durable artifact and the implementation as disposable. Regenerating the implementation specializes it to each workload and device. However, that work lives in a forgiving setting: a component's externally visible effects, its writes and device commands, are recoverable, and the generator is honest, so trust is discharged by re-execution. We target the unforgiving setting: systems code whose effects are irreversible, produced by a generator that may be adversarial. There, re-execution cannot check an effect after the fact, and a proof fails silently when its assumptions do. We take the position that the only safe way to operate here is to deny the generated code the authority to act. The generated code only plans, while a fixed trusted mediator owns every effect and performs one only when the specification would have produced it. Because the guarantee lives in the mediator, not the code, it survives regeneration. We instantiate this as a reference monitor for regenerated device drivers, and characterize the *mediability envelope*, six conditions on the effect vocabulary: legibility, spec-input observability, correlatability, completeness, outcome enumerability, and explicit durability. They decide when such mediation is possible.

Keywords: generated systems code, reference monitors, runtime refinement, effect authority, device drivers

ACM Reference Format:
Jinhao Hu, Ashvin Goel, and Laurent Bindschaedler. 2026. Don't Trust the Code, Check Its Effects: Runtime Refinement for Regenerated Systems Code Under an Adversarial Generator. In *Practical Adoption Challenges of ML for Systems (PACMI '26), September 29–October 2, 2026, Prague, Czech Republic.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3843967.3844678

PACMI '26, Prague, Czech Republic

ACM ISBN 979-8-4007-2998-0/26/09
https://doi.org/10.1145/3843967.3844678

1 Generating Systems Components

A growing body of work uses large language models to regenerate systems code on demand, treating the specification, not the implementation, as the durable engineering artifact [4, 5, 27, 30, 32, 42]. The pattern already spans the stack: file systems synthesized from specifications and checked against specification-derived test suites [26], distributed protocols generated with mechanized proofs [2], and analytical database engines synthesized per workload for order-of-magnitude speedups [46]. A deployer curates a specification, a model emits a native implementation, and regenerates it rather than patching it when the specification changes or a defect surfaces. The specification is what humans maintain, and the implementation is disposable.

Regeneration buys specialization. Systems components must be tuned to hardware and workload, and a single implementation commits once to a large design space: a storage engine's layout, a network stack's congestion control, a driver's queueing strategy. A model can search that space and re-search it per deployment. The deployer re-specializes by regenerating, never by editing.

These systems share a setting that makes them tractable: their effects are recoverable, an analytical query can be rerun, an engine re-benchmarked, a wrong output recomputed, and their generators are treated as honest. Trust is therefore discharged by *re-execution*: validate the implementation against tests or a workload and regenerate on failure [26, 46]. This works precisely because every effect can be undone.

We target the unforgiving setting, where that assumption fails. Systems code produces irreversible, externally visible *effects*, the operations through which it changes state outside itself, such as a device command, a packet send, or a disk write. Once an effect reaches the host it cannot be recalled, so re-execution cannot check it after the fact. And the generator cannot be assumed honest: a compromised provider, a poisoned training set, or prompt injection can emit an implementation that issues an out-of-spec effect [8, 15]. Under irreversible effects and an adversarial generator, validation by re-execution is unavailable, and, as we argue next, so is verification alone.

A device driver is the sharpest instance, and our running example. A model generates the native driver that turns kernel block requests into device commands, managing queues, DMA, and interrupts, and regenerates it per deployment.

Drivers are a dominant share of kernel code and its faults, tuned across thousands of devices, and their most dangerous effect, a DMA into arbitrary host memory, is irreversible and unconfined by construction.

2 From Verification to Enforcement

Section 1 established the setting; we now pin the threat model. We treat the generated component as *Byzantine*: it may compute arbitrarily, hide an out-of-spec effect behind a branch no test exercises, and misreport anything it tells us. Every value the code supplies is untrusted.

The natural response is to verify the code. Prove that the generated component refines its specification, admit it, and run it with authority over the device, as in proof-carrying generation and admission-time regeneration [2, 3, 28, 48]. Trust is discharged once, before running, as a property of the artifact.

Verification alone does not survive this setting, because its guarantee is conditional on assumptions the setting breaks. A proof yields safety only if the generator produces a checkable proof for every regeneration, the host model it assumes is complete and sound, the toolchain from proof to binary is trusted, and the running bits are bound to the verified artifact. An adversarial generator need not prove anything; a model of real hardware is never complete; compilers and extraction are rarely verified; and regenerated native code is rarely bound to a proof. Where any assumption breaks, the proof fails silently and the component commits an irreversible effect anyway. Verification establishes a property of a *description*; what reaches the device is a property of the *run*, and under regeneration the two need not coincide.

Our position is different: trusting the code is not enough, so we deny the generated code the authority to act in the first place. Safety then no longer depends on its correctness. The generated component becomes a *planner* that proposes effects but performs none; a fixed, handwritten *mediator* owns every *effect primitive*, the low-level operations through which state leaves the machine, and performs one only after checking, against the specification and its own state, that the effect is one the specification would produce. The guarantee no longer rests on the code but on the mediator, which observes the actual effect at the one point it can still be refused, from a bounded abstract state, not the device's internals.

This trades a verification toolchain for a mediator that carries its own abstract state, its own account of what is safe, and its own compiler. We do not claim that the trusted base is smaller, and size is not the useful comparison. What separates them is the work a regeneration forces. A proof binds one binary, so every regenerated implementation must earn its evidence again, from proof through extraction to binary. Our compiler checks the specification, which does not change when the implementation does, so a regenerated driver re-runs nothing and gains no authority.

This is not a claim that verification cannot be safe; a stack verified down to a trusted machine model needs no mediator. Under an adversarial generator, native code, and irreversible effects, though, verification's preconditions are unmet, so the trusted decision must be made at the effect, not the artifact. Where achievable, verification raises availability by making proposals usually correct, but safety rests on enforcement.

We make three contributions. We recast trust for regenerated systems code as *effect authority* rather than code verification: the code only plans, and a fixed mediator owns and adjudicates every effect. We instantiate this as a reference monitor for regenerated drivers (Section 3), extending driver-safety monitoring [47] to functional refinement. And we characterize the *mediability envelope* (Section 4): six conditions on the effect vocabulary that decide when such mediation is possible. This is a position paper: no formal semantics, mechanized proof, implementation, or measurement.

3 The Boundary, by Example

Figure 1 illustrates the boundary we call *proposal-commit separation*: the driver *proposes* device commands, and the mediator commits one only when it refines the specification.

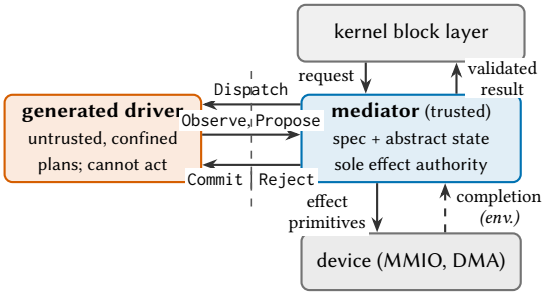

Figure 1. Proposal-commit boundary for a driver. The kernel's request enters the mediator, which dispatches to the confined driver; the driver may only `Observe` and `Propose`. The mediator alone holds the specification and effect primitives (MMIO, DMA), committing only effects that refine it.

The specification. The mediator holds a specification of the driver's *interface*, not of its implementation. Figure 2 gives the `WRITE` clause: a small, device-generic transition relation over the outstanding requests and the queue state. It is a fraction of the driver's queueing and DMA machinery, so checking a command against it costs far less than producing one. Runtime values are taken from the request itself. `req.lba` names whatever the kernel asked for in the request now pending, and the mediator, which holds that request, rederives the value rather than taking the driver's word for it. The driver chooses only the queue slot, and the specification bounds that choice to the slots the driver observes free.

The protocol. The driver never touches the device. The mediator `Dispatch`es a pending request. While planning, the driver must first `Observe` the abstract state the specification permits it to see, the free queue slots and DMA grants of

```
operation WRITE(lba, len, buf)  -- kernel's request
  observe  queue.free, dma.grant[buf]
  permit   op   = WRITE        -- fixed by request
           lba  = req.lba
           len  = req.len
           dma  = req.buf
           slot in queue.free  -- driver's choice
  outcome  ok       -> req.done
           EIO      -> req.failed, device unchanged
           short(n) -> req.partial(n)
```

Figure 2. The `WRITE` clause of a driver specification. The mediator's abstract state is the outstanding requests, the queue's free slots, and the DMA grants. `permit` separates the fields the specification fixes from the pending request, which the mediator re-derives, from the one field the driver is free to choose. `outcome` enumerates what the primitive may return and the abstract-state transition each produces.

Figure 2; it may observe again before proposing. `Observe` returns state, never authority, together with a fresh *sealed token*, a value the driver cannot forge, binding the request, the versions of the abstract state that were read, and an expiry. The driver necessarily plans against a snapshot, and that snapshot may move before it proposes; the mediator rechecks the token during `Validate`, under the commit lock. A later observation invalidates the preceding token, so a proposal always describes a plan against one identified observation.

`Propose` is the driver's only message that asks the mediator to perform an effect. It submits an *attempted effect*, a device command, and nothing more, together with its token. Every field *of the proposal* is an untrusted witness. The driver says nothing about what the effect will return, because the specification has already said it: Figure 2 closes the outcome set at three entries for `WRITE` and pairs each one with its abstract-state transition. Once the primitive has run, the mediator therefore needs nothing further from the driver.

The mediator runs `Validate` under a commit lock, and rejects a replayed, expired, superseded, or stale token. It then checks that every field the specification fixes matches what it re-derives from the pending request, which it holds itself and never reads from the proposal, and that each freely chosen field falls inside the permitted set. `Validate` consumes the token whether it accepts or rejects the proposal: one request cannot produce a second effect by replaying a token.

Only then does the mediator `Commit`. It invokes the trusted primitive and reads the *actual* outcome, which Figure 1 shows returning as *completion*. The specification, not the driver, says which transition that outcome produces, and the mediator records it and delivers the validated result to the kernel. A device error that the specification lists, such as the `EIO` of Figure 2, is a committed outcome rather than a failure of mediation. The mediator records `req.failed` and reports the error upward. When `Validate` fails instead, it `Rejects` before any irreversible primitive runs, so a rejected proposal is externally silent.

Catching a wrong write. The kernel issues `write LBA 100..107 from` B; consider a driver that proposes `{op=WRITE, LBA=200, len=8, src=`B`, slot=`k`}` instead of the requested LBA 100: it overwrites the wrong block with B's contents, an effect no later regeneration can undo. Every other check passes: the DMA source is still the authorized buffer B, so an IOMMU, which never sees an LBA at all, has nothing to object to; 200 is a legal LBA, so a device-safety monitor that never observes the kernel's request, such as a reference validation mechanism [47], has no requested value to compare against; an LBA the test suite never exercised would pass sampled admission validation [26] just the same. Only a check that holds the pending request and compares it against the proposed command catches this: the mediator's `Validate` rejects it before the doorbell rings, because $100 \neq 200$, and nothing is written to the device. This same comparison covers the rest of the proposal. A wrong `len`, or a DMA source other than B, fails identically, since Figure 2 fixes both to the request. A `slot` outside the observed free set fails instead against the permitted set, and a proposal naming no pending request is rejected at the token. Every `Propose` carries the token from its most recent `Observe`, and `Validate` consumes it, so a request in flight yields at most one effect, however often the driver replans, and a driver that proposes nothing stays silent. Absent a crash, one request never yields a second effect.

Confinement and ownership. The driver runs with no direct kernel access, no device register mapping, and no DMA authority; the mediator is its only outward channel. Confinement blocks any effect the driver might attempt on its own, rather than relying on inspect-and-continue `seccomp` notification, which is unsafe against concurrently mutated arguments [21]. The mediator owns the trusted effect primitives, observation-token issuance, DMA authorization through the IOMMU, the abstract state of Figure 2 and the commits that advance it, the append-only log in which each committed effect is recorded together with the outcome it produced, and the specification interpreter. It is the authority for the declared effect vocabulary.

Concurrency. Drivers may plan and compute concurrently, but the mediator serializes the parts that matter: validation, primitive invocation, abstract-state update, and log insertion run under a single commit lock, tokens are revalidated after the lock is taken, and a stale proposal is rejected and replanned. The commit-log order is the specification's serialization order. Serializing primitives does not make a multi-primitive guest operation atomic; an operation that needs cross-effect atomicity requires one atomic primitive, substrate locking, or exclusion from the vocabulary.

Guarantee. A realization should claim *committed-trace safety*: every committed sequence of externally visible effects is a prefix of a trace the specification allows over the

declared effect vocabulary. It presupposes that the effect vocabulary lies within the mediability envelope (Section 4), and it is conditional on a correct specification and interpreter, complete mediation with no bypass, trusted effect primitives, serialized commits, outcome enumerability, and a stated crash model. The guarantee excludes liveness and availability, since a Byzantine driver may propose nothing or select the worst permitted proposal, and it does not cover timing channels or defects in the specification itself.

4 The Mediability Envelope

A specification is written against an interface: it names that interface's operations, such as `mkdir(path,mode)` or `write LBA 100..107`, and the objects those operations touch, such as paths, credentials, queue slots, and DMA buffers. We call that interface's level of abstraction the specification's *altitude*. The architecture works only where the effect vocabulary lets the mediator see and check the right thing cheaply, at that altitude, without becoming the code it checks. Refinement is a relation over what the mediator must observe and hold against the specification across the life of a request: the *intent* (the requested operation), the *delta* (what actually changed), the *result* (what the caller is told), and, over time, whether what was recorded survives a crash. Whether a vocabulary supplies these, cheaply and independently, defines a *mediability envelope*. Six conditions delimit it, and they are of two kinds. The first three ask what the effect vocabulary supplies, and altitude settles them, so they decide whether a domain can be mediated at all. The last three ask what the deployment supplies, and engineering settles them at any altitude. We read each condition against two vocabularies: the driver's device commands, which satisfy it, and a file system's block writes, which serve as a foil for what failure looks like. Where a condition fails, the mediator can regain it only by reconstructing what the vocabulary discarded.

What the vocabulary must supply. *Legibility.* A single effect must be legible, on its own, as an instance of one operation: from the effect alone, without replaying what came before, the mediator must be able to tell which operation it belongs to. A device command names its operation: Figure 2's `WRITE` is legible on its own, distinguishable from a `READ` by its own encoding. A block write does not: the same write to a directory block and a bitmap could be a `mkdir`, an `rmdir`, a rename, or a corruption, so whether an operation changed state, changed nothing, or did the opposite is invisible. Ambiguity denies the mediator the intent.

Spec-input observability. The mediator must observe every input the transition reads. Figure 2's `WRITE` observes `dma.grant[buf]`, the caller's own authorized source buffer, before permitting a `slot`; `mkdir` similarly is licensed only against the caller's credentials, but the virtual-filesystem interface carries them while the block layer strips them, so a permission-dependent transition cannot be checked there at any cost.

Correlatability. Effects must group into per-operation transactions with a recoverable order under concurrency. At the specification's altitude a device command is self-delimiting, so grouping is free: one proposal, one transaction, serialized at commit. Figure 2's `WRITE` is exactly this. Block writes are not: concurrent operations interleave unlabeled writes over shared structures, so they cannot be grouped by address, and no serialization recovers an identity the vocabulary never carried. Even where effects self-delimit, physical write order can diverge from commit order, and only the commit lock recovers it. Certifying a stream that fails either half means modeling the component's own concurrency control.

What the deployment must supply. *Completeness.* With intent legible, the record of what changed must be kept sound: every externally visible state change must cross the boundary, as a validated effect primitive or a mediator-owned environment transition; any third path voids the guarantee. The standing risk is effect authority the component already holds: a writable shared mapping or DMA lets it change host-visible state before any proposal. Both must be excluded or reified into proposals, shared memory through shadow pages, DMA through IOMMU-authorized buffers. Figure 2's `dma.grant[buf]` is exactly this reification, checked before any `WRITE` is permitted, not assumed.

Outcome enumerability. The record must also be trustworthy on the way out: the outcomes a primitive may produce, as reported at the mediator's interface, must be closed and enumerable, so that whatever it returns, success, a device error, or a partial completion, can be classified against the specification's allowed set the moment it returns, before any abstract state is advanced, as Figure 2's three-entry `outcome` clause does for `WRITE`. The mediator needs no model of why the device produced an outcome, only a complete enumeration of the forms it can take; an unenumerable outcome space forces the mediator to guess at commit time what an incomplete device model forces a prover to guess at admission time.

Explicit durability. Persistence extends the same concern into time: whether what the record says happened still holds after a crash. Figure 2's `WRITE` illustrates the gap: its `ok` outcome means the bytes reached the device, not that they survive a crash. Durability must be a first-class effect the component proposes, a flush or barrier, not a property that emerges from write ordering, since a crash truncates the stream at an arbitrary point while destroying volatile state. If it is explicit, the mediator checks that a claimed-durable `WRITE` issued a following flush the device confirmed; if implicit, the mediator must reconstruct the component's barrier discipline and keep its own crash-consistent store.

One condition straddles the split. Correlatability's grouping half belongs to the vocabulary, but its ordering half is

closed only by the commit lock. So the rule narrows: mediate at the specification's own interface, never below it, and treat confinement, serialization, outcome closure, and durability as separate engineering. Below that altitude the mediator must *lift* effects back to operations, re-implementing the read path, concurrency control, and recovery, and re-attaching the intent and authority the vocabulary discarded.

Within the envelope, a separate question is economic: how much standing state a check must consult. A driver sits at the cheap end (queue descriptors, DMA grants); a file system at the virtual-filesystem interface sits at the costly end, owning the namespace and per-file digests while the component owns the physical data plane; arbitrary specialization or bugs then degrade availability, not silently corrupt data. At the block layer the same system falls out of the envelope: legibility, spec-input observability, and correlatability fail. Altitude is the line between checking and re-implementing.

5 Related Work

The approaches we reject have deep lineages. Table 1 organizes the design space along two questions: when the check runs, at admission or at commit, and what it checks, permission or correctness. *Proof* certifies the artifact at admission, before it runs, from proof-carrying code [34] to verified systems [22, 23] and provers aimed at generated code [2, 3, 28, 48]. The proof is re-earned for every regeneration. *Policy* withholds effect authority and asks only whether an effect is permitted: the eBPF verifier [14] checks at admission, while software fault isolation [43] and NaCl [49] check at commit. Policy substrates such as [1, 44, 50] can also interpose at that point, but typically decide whether a primitive, fd, address, or DMA buffer is authorized, not whether it refines the current trusted request and contract state; a policy that retains and checks the specification's transition relation there is already a mediator in this paper's sense.

Our mediator composes established mechanisms: confined code that delegates to a trusted agent [13, 21], opaque handles from capability systems [16, 41, 45], speculate-and-commit from system transactions [20, 37], and behavioral-model checking from model-carrying code [40], edit automata [25, 39], and shield synthesis [6]. Runtime verification compiles specifications into monitors that run alongside the program [7, 24], and runtime refinement checking is two decades old [9], but both observe a trace rather than gating it. Output-buffering speculation defers an external effect until it is known safe [35, 36], but it presumes a recoverable substitute stands in for the effect; it cannot withdraw a command the device has already run.

The closest work is driver safety through a reference validation mechanism [47]: untrusted, restartable native drivers, each interaction checked at commit by a specification-compiled monitor that survives replacement. That monitor enforces a device-safety automaton; ours checks functional refinement against a generator that rewrites the driver on demand. Runtime checking of file-system metadata at commit is a similar line: Recon and its successor check consistency and atomicity invariants over block writes [11, 12], but below the specification's altitude, reading only metadata and never the caller's request, so they catch structural corruption, not permitted-but-wrong outcomes. Concurrent work mediates LLM *agents* at the tool boundary [8, 10, 17, 18, 31, 33, 38], but assumes an honest generator and acts above regenerated native systems code. Our companion workshop papers place a mediation boundary around agent explorations [19] and price fleets' residual risk [29], but target agent workloads, not regenerated systems code.

	Permission (effect is allowed)	**Correctness** (effect refines the spec)
At admission per artifact; redone on regeneration	static allowlists; the eBPF verifier	proof-bearing generation (PCC, Verus, IDS); admission validation (SYSSPEC)
At commit per effect; survives regeneration	policy substrates (NaCl, gVisor, WASI, seccomp); device-safety monitors (RVM)	**runtime refinement (this paper)**

Table 1. The design space for trusting regenerated systems code. Admission-time checks attach to the artifact and are redone on every regeneration; commit-time checks live in the mediator and survive it. Prior work occupies three cells; this paper claims the fourth.

6 What a Realization Must Answer

A realization must answer four engineering questions beyond mediability. *Cost*: how large is the mediator's trusted code and abstract state against the component it guards, what latency does serialized commit add, and how often do proposals go stale under concurrency? If the mediator grows with the implementation, the envelope was drawn at the wrong altitude. Whether drivers clear that bar is a fair doubt: the per-effect check is cheap only because the driver's abstract state stays small, and thin margins make the case for regenerating device by device. *Adapter trust*: each effect primitive reaches the device through an *adapter*, the small piece of trusted code that turns a validated command into the register writes and DMA descriptors the hardware expects. Is each adapter mechanically verified, audited against a contract, or excluded from the trusted base, so the system claims a mechanized core with selected verified adapters, not a verified mediator? *Availability*: a Byzantine component can propose nothing, stall with fresh observations, or exhaust its permitted choices, so a deployment needs a deadline, budgets, or a fallback. *Unknown unknowns*: a defect in the specification is enforced faithfully, and an unenumerated outcome is an outcome-enumerability failure, so a realization must fail stop on both.

Regenerated systems code needs a trust boundary that survives regeneration. Denying it effect authority and mediating each effect provides one, and the envelope says when: this paper isolates the laws a regenerable OS builds on.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. USENIX Association, 419–434.

[2] Shubham Agarwal, Alexander Krentsel, Shu Liu, Mert Cemri, Audrey Cheng, Rui Meng, Tomas Pfister, Chun-Liang Li, Sylvia Ratnasamy, Aditya Parameswaran, Matei Zaharia, Ion Stoica, and Mohsen Lesani. 2026. Inductive Deductive Synthesis: Enabling AI to Generate Formally Verified Systems. *CoRR* abs/2605.23109 (2026). arXiv:2605.23109 [cs.AI]

[3] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. 2025. AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement. In *Proceedings of the 42nd International Conference on Machine Learning (ICML '25) (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 587–615.

[4] Thomas Anderson, Ratul Mahajan, Simon Peter, and Luke Zettlemoyer. 2025. *Self-Defining Systems*. White paper. Paul G. Allen School of Computer Science & Engineering, University of Washington. https://self-defining-systems.cs.washington.edu/papers/sds.pdf

[5] Laurent Bindschaedler. 2026. ForkOps: Operating Fleets of Generated Software Variants. In *Proceedings of the 1st International Workshop on Software Genomics (SWGeno '26)*. Munich, Germany. Co-located with ASE 2026.

[6] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. 2015. Shield Synthesis: Runtime Enforcement for Reactive Systems. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '15) (Lecture Notes in Computer Science, Vol. 9035)*. Springer, 533–548. doi:10.1007/978-3-662-46681-0_51

[7] Feng Chen and Grigore Roşu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '07)*. Association for Computing Machinery, 569–588. doi:10.1145/1297027.1297069

[8] Zheng Chen, Hanqing Liu, Duling Xu, Dong Dong, Jialin Li, Bangzheng Pu, and Jidong Zhai. 2026. Cordon: Semantic Transactions for Tool-Using LLM Agents. *CoRR* abs/2606.17573 (2026). arXiv:2606.17573 [cs.OS]

[9] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. 2005. VYRD: VerifYing Concurrent Programs by Runtime Refinement-Violation Detection. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, 27–37. doi:10.1145/1065010.1065015

[10] Marcelo Fernandez. 2026. Agent Control Protocol: Admission Control for Agent Actions. *CoRR* abs/2603.18829 (2026). arXiv:2603.18829 [cs.CR]

[11] Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. 2014. Checking the Integrity of Transactional Mechanisms. *ACM Transactions on Storage* 10, 4, Article 17 (2014), 23 pages. doi:10.1145/2675113

[12] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying File System Consistency at Runtime. *ACM Transactions on Storage* 8, 4, Article 15 (2012), 29 pages. doi:10.1145/2385603.2385608

[13] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*. Internet Society, 187–201.

[14] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. Association for Computing Machinery, 1069–1084. doi:10.1145/3314221.3314590

[15] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec '23)*. Association for Computing Machinery, 79–90. doi:10.1145/3605764.3623985

[16] Norman Hardy. 1985. KeyKOS Architecture. *ACM SIGOPS Operating Systems Review* 19, 4 (1985), 8–25. doi:10.1145/858336.858337

[17] Jun He and Deying Yu. 2026. Protocol-Driven Development: Governing Generated Software Through Invariants and Continuous Evidence. *CoRR* abs/2605.12981 (2026). arXiv:2605.12981 [cs.SE]

[18] Jun He and Deying Yu. 2026. Sovereign Assurance Boundary: Certificate-Bound Admission for Agentic Infrastructure. *CoRR* abs/2606.11632 (2026). arXiv:2606.11632 [cs.CR]

[19] Jinhao Hu, Bardia Mohammadi, Ashvin Goel, and Laurent Bindschaedler. 2026. Externalization Barriers: An OS Abstraction for Untrusted Agent Exploration. In *1st Workshop on Operating Systems for Agents (AgenticOS '26)*. https://os-for-agent.github.io Co-located with SOSP 2026.

[20] Suman Jana, Donald E. Porter, and Vitaly Shmatikov. 2011. TxBox: Building Secure, Efficient Sandboxes with System Transactions. In *2011 IEEE Symposium on Security and Privacy (S&P '11)*. IEEE Computer Society, 329–344. doi:10.1109/SP.2011.33

[21] Michael Kerrisk. 2026. `seccomp_unotify(2)` — *Linux manual page* (6.18 ed.). Linux man-pages project. https://man7.org/linux/man-pages/man2/seccomp_unotify.2.html

[22] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, 207–220. doi:10.1145/1629575.1629596

[23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1, Article 85 (2023), 30 pages. doi:10.1145/3586037

[24] Martin Leucker and Christian Schallhart. 2009. A Brief Account of Runtime Verification. *Journal of Logic and Algebraic Programming* 78, 5 (2009), 293–303. doi:10.1016/j.jlap.2008.08.004

[25] Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *International Journal of Information Security* 4, 1–2 (2005), 2–16. doi:10.1007/s10207-004-0046-8

[26] Qingyuan Liu, Mo Zou, Hengbin Zhang, Dong Du, Yubin Xia, and Haibo Chen. 2026. Sharpen the Spec, Cut the Code: A Case for Generative File System with SYSSPEC. In *Proceedings of the 24th USENIX Conference on File and Storage Technologies (FAST '26)*. USENIX Association, 291–311.

[27] Shu Liu, Alexander Krentsel, Shubham Agarwal, Mert Cemri, Ziming Mao, Soujanya Ponnapalli, Alexandros G. Dimakis, Sylvia Ratnasamy, Matei Zaharia, Aditya Parameswaran, and Ion Stoica. 2026. The Time is Here for Just-in-Time Systems: Challenges and Opportunities. *CoRR* abs/2605.24096 (2026). arXiv:2605.24096 [cs.DB]

[28] Yuwei Liu, Xinyi Wan, Yanhao Wang, Minghua Wang, Lin Huang, and Tao Wei. 2026. KVerus: Scalable and Resilient Formal Verification Proof Generation for Rust Code. *CoRR* abs/2605.03822 (2026). arXiv:2605.03822 [cs.SE]

[29] Bardia Mohammadi and Laurent Bindschaedler. 2026. The Irreversibility Budget: Fleet-Level Risk Accounting and Admission Control for Agent Operating Systems. In *1st Workshop on Operating Systems for Agents (AgenticOS '26)*. https://os-for-agent.github.io Co-located with

SOSP 2026.

[30] Bardia Mohammadi, Lars Klein, Aman Chadha, Akhil Arora, and Laurent Bindschaedler. 2026. The Working Set of a Coding Agent: Coherence Debt in Repository-Scale Tasks. *CoRR* abs/2608.16630 (2026). arXiv:2608.16630 doi:10.48550/arXiv.2608.16630

[31] Bardia Mohammadi, Nearchos Potamitis, Lars Klein, Akhil Arora, and Laurent Bindschaedler. 2026. Atomix: Timely, Transactional Tool Use for Reliable Agentic Workflows. *CoRR* abs/2602.14849 (2026). arXiv:2602.14849 doi:10.48550/arXiv.2602.14849

[32] Martin Monperrus. 2026. Bootstrapping Coding Agents: The Specification Is the Program. *IEEE Software* 43, 4 (2026), 19–22. doi:10.1109/MS.2026.3687051

[33] Royce Moon and Lav R. Varshney. 2026. Containment Verification: AI Safety Guarantees Independent of Alignment. In *Second Workshop on Agents in the Wild: Safety, Security, and Beyond at ICML 2026*. OpenReview.net. https://openreview.net/forum?id=NGNK03iIam

[34] George C. Necula. 1997. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. Association for Computing Machinery, 106–119. doi:10.1145/263699.263712

[35] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. 2005. Speculative Execution in a Distributed File System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*. Association for Computing Machinery, 191–205. doi:10.1145/1095810.1095829

[36] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, 1–14.

[37] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, 161–176. doi:10.1145/1629575.1629591

[38] Igor Santos-Grueiro. 2026. Lingering Authority: Revocable Resource-and-Effect Capabilities for Coding Agents. *CoRR* abs/2606.22504 (2026). arXiv:2606.22504 [cs.CR]

[39] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Transactions on Information and System Security* 3, 1 (2000), 30–50. doi:10.1145/353323.353382

[40] R. Sekar, V. N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. 2003. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Association for Computing Machinery, 15–28. doi:10.1145/945445.945448

[41] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*. Association for Computing Machinery, 170–185. doi:10.1145/319151.319163

[42] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. 2024. Specifications: The missing link to making the development of LLM systems an engineering discipline. *CoRR* abs/2412.05299 (2024). arXiv:2412.05299 [cs.SE]

[43] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*. Association for Computing Machinery, 203–216. doi:10.1145/168619.168635

[44] WASI Subgroup. 2026. WASI 0.3.0. https://github.com/WebAssembly/WASI/releases/tag/v0.3.0 Ratified standards-track release, 11 June 2026.

[45] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security '10)*. USENIX Association, 29–46.

[46] Johannes Wehrstein, Timo Eckmann, Matthias Jasny, and Carsten Binnig. 2026. Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines. *CoRR* abs/2603.02001 (2026). arXiv:2603.02001 [cs.DB]

[47] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. 2008. Device Driver Safety Through a Reference Validation Mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, 241–254.

[48] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2, Article 396 (2025), 29 pages. doi:10.1145/3763174

[49] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy (S&P '09)*. IEEE Computer Society, 79–93. doi:10.1109/SP.2009.25

[50] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19)*. USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/young