ONE GOAL, MANY COMMANDS: CHARACTERIZING DENYLIST FRAGILITY IN AI AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

The adoption of AI agents is increasing rapidly. Terminal AI agents, i.e., AI agents that run in terminal environments, are widely used. Terminal AI agents rely heavily on shell commands to interact with host systems. They adopt denylists to mitigate the security risks introduced by executing arbitrary commands, which serve as the last line of defense against dangerous and harmful operations. However, denylists can be fragile, as the same goal (e.g., performing a harmful operation) can often be achieved by many commands. This paper presents the first systematic characterization of command denylist fragility in terminal AI agents. The paper formalizes the command denylist fragility problem and proposes an LLM-driven pipeline, SHELLSIEVE, to detect such fragility. It prompts the LLM to propose possible bypasses and iteratively repairs them using feedback from a validator that executes them in a sandbox. In the evaluation, we applied SHELLSIEVE to 4,668 real-world command denylists (containing 36,846 denylist rules) collected from GitHub. The evaluation shows several key findings, including that 69.5–99.8% of the denylists are fragile, that this fragility occurs consistently across projects and agents, and that one cause of this fragility stems from denylist authors overlooking less-known commands. Our pipeline and findings will hopefully facilitate future research and practice regarding the command denylists used by AI agents.

1 INTRODUCTION

AI agents running in terminal environments, such as Claude Code and Codex, have recently emerged as versatile personal assistants (Merrill et al., 2026). One of the most important capabilities of these *terminal AI agents* is their ability to dynamically synthesize and execute shell commands (Liu et al., 2026). However, this capability creates a *dilemma between flexibility and security*: it grants the agents broad autonomy to finish tasks with minimal human intervention, while security may be compromised by the extensive permissions required. Currently, the common practice to mitigate the dilemma is to gate commands using three lists: a denylist, an allowlist, and an implicitly defined ask-list that relies on an evaluator to decide on everything else. Among the three lists, the denylist is the last line of defense against inherently dangerous commands.

The problem. However, denylists can be fragile, as *the same goal (e.g., performing a harmful operation) can often be achieved by many commands*. Denylists may block some commands that perform an operation, but overlook others that bypass their protection. Even the denylist shipped with a widely used agent by default and well maintained by its developers can overlook bypass commands that compromise its protection, let alone the denylists compiled by regular users of these agents (see §2). Therefore, the evaluation and characterization of this fragility is an urgent and critical need for future research and practice.

```
"deny": [
    "cat *.env",
    "head *.env",
    "tail *.env",
    "less *.env",
    "more *.env",
    "grep *.env"
]
```

Listing 1: A Claude Code command denylist. "*" is a wildcard.

Threat model. We assume that the adversary can manipulate the AI model in the agent to propose executing any command. This can be done via prompt injection or similar techniques (Zhan et al., 2024; Khan, 2026; Liu et al., 2026), but cannot run a shell directly on the host, as command gating would be moot otherwise. We also assume that the adversary knows

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

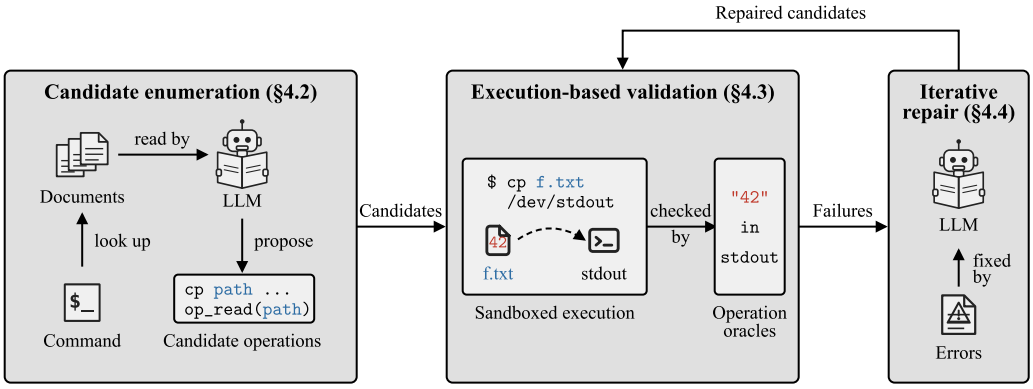


Figure 1: End-to-end SHELLSIEVE pipeline. Each input (a command with arguments) flows through three stages. The *candidate enumerator*, an LLM conditioned on `man` pages, `--help` outputs, and GTFOBins entries, proposes candidate operations that the command may perform. The *validator* then executes the command against operation-specific oracles to decide whether these candidates are correct. Failures are routed through the *repair loop* back to the LLM with diagnostic messages, for at most M rounds. The confirmed operations are the pipeline's output.

the gating policy in full, as these lists are typically stored alongside the source code in a Git repository (Anthropic, 2026b; OpenAI, 2026b; OpenCode, 2026a).

Challenges and methodology. To achieve such characterization, we must identify operations that a denylist blocks and bypass commands that compromise a denylist to perform the operations, which is difficult. Thus, we propose SHELLSIEVE, a pipeline with two novel designs (see Figure 1) for this purpose. First, it leverages an LLM to propose candidate operations and bypass commands. LLMs' expert-level capability to synthesize shell commands (Sladić et al., 2024) enables them to produce thousands of diverse, high-quality candidates that no traditional technique can propose. Second, SHELLSIEVE defines and validates candidate operations and bypass commands based on the side effects they leave in a sandbox environment after execution. Thus, it avoids ambiguity and false positives in the characterization.

Related work Prompt injection has been shown to hijack coding agents into running attacker-chosen commands (Zhan et al., 2024; Liu et al., 2026; Khan, 2026), leading to the threat model we apply. Command gating has been engineered into agents and agent sandboxes (Anthropic, 2026b; OpenAI, 2026b; OpenCode, 2026a; Agentsh Project, 2026). To our knowledge, its fragility has never been measured. Living-off-the-land catalogs such as GTFOBins (GTFOBins Project, 2026) enumerate abusable binaries by hand, yet they consist of fixed proof-of-concept examples and cannot directly guide the search for bypasses in concrete contexts. Static and symbolic analyses could, in principle, enumerate what a binary does (Gosain & Sharma, 2015; Wong et al., 2015; Pandita et al., 2012), but have scalability issues across hundreds of heterogeneous binaries on a host.

Findings. We evaluate 4,668 real-world AI agent denylists (containing 36,846 denylist rules) collected from GitHub with SHELLSIEVE and reveal several key findings:

- First, the denylist fragility problem is severe. 69.5–99.8% of denylists cannot fully block operations they aim to block.
- Second, the severity of the fragile denylist problem is consistently high across projects and agents. The proportion of denylists that overlook bypass commands never falls below 50.0% across projects receiving varying levels of attention. Similarly, the proportion never falls below 37.2% for projects using different agents.
- Third, we statistically confirmed a root cause of the denylist fragility problem: the authors of denylists ignore less-known commands, creating bypasses. By contrast, multi-purpose commands (e.g., interpreters) deliberately unlisted from the denylists for their benign use do not explain the problem. Even when they are excluded, 68.2–99.8% of denylists still overlook bypass commands.
- Fourth, fixing fragile denylists will require adding numerous commands (averaging 334 if they aim to block specific operations) to complete them, which is a huge burden for anyone who wants to resolve the problem.

2 BACKGROUND

Three-list command gating. An agent is useful because it can run whatever commands a task requires, yet the same freedom lets an adversary who hijacks it perform arbitrary, harmful operations on the host. The common practice to mitigate this tension between flexibility and security is to gate the shell commands before execution using three lists, as illustrated in Figure 2:

- **Denylist.** Denylists block obviously dangerous operations that the agent must never run, such as deleting system files or leaking secrets. Listing 1 shows an example denylist from a GitHub project that uses Claude Code.
- **Allowlist.** Allowlists permit benign operations that the agent may run without any intervention, such as reading a non-confidential file.
- **Ask-list.** Everything not matched by the allowlist or the denylist goes into the ask-list and is delegated to an authoritative evaluator, typically the user, for a final decision (Anthropic, 2026b; OpenAI, 2026b; OpenCode, 2026a; Agentsh Project, 2026).

This basic three-list command-gating architecture is observed in various terminal AI agents, including Claude Code (Anthropic, 2026a), Codex (OpenAI, 2026a), and OpenCode (OpenCode, 2026b). Implementations across different agents may vary in certain details, though.

Among the three lists, the denylist is the last line of defense against malicious commands, given that the effectiveness of the ask-list is largely compromised by *approval fatigue*, which leads 93% of users to blindly accept approval requests (Anthropic, 2026a). Adopting an AI judge to make the decision on behalf of the user mitigates the problem, but it may encounter LLM-specific issues such as hallucinations (Chrysos et al., 2025) and adversarial suffixes (Zou et al., 2023).

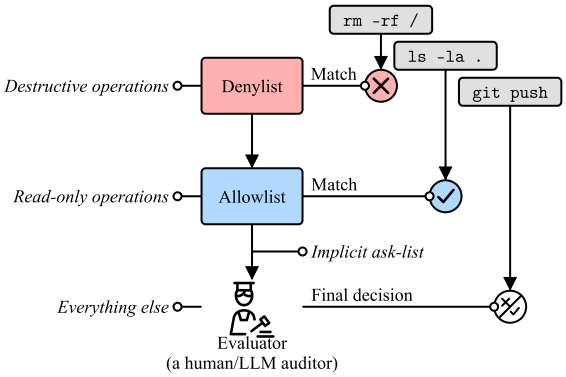


Figure 2: A three-list command-gating model. Implementations may vary in some details.

Denylist fragility. However, denylists can be fragile, as they often overlook less common commands that can perform the same operations as the commands they block. These overlooked commands *bypass* the denylists' protection against those operations. Even denylists well maintained by the agents' developers can be fragile, let alone those hurriedly compiled by ordinary users. We have found bypasses for the built-in denylist of Claude code and for denylists included in open-source projects on GitHub. Appx. A presents the bypasses we found.

3 PROBLEM FORMALIZATION

This section formalizes the problem of fragile command denylists. The definitions in this section set the objective for the rest of this paper.

3.1 DENYLISTS AND DENY-RULES

Definition 1 provides the formal definitions of denylists and denylist rules (or deny-rules for short).

Definition 1 (Deny-rules and denylists) *A deny-rule r is a predicate over a command with its argument list:*

$$r : C \times A^n \to \{\top, \bot\}$$

where C is the set of all available commands, A^n is the set of all possible argument lists. r blocks a command c with argument list a, denoted as $r \rhd c, a$, if

$$r\,(c, a) = \top$$

A denylist L is a set of deny-rules:

$$L \in \mathcal{P}(R)$$

where R is all possible deny-rules, and $\mathcal{P}(\cdot)$ denotes power sets. L blocks a command c with argument list a, denoted as $L \rhd c, a$, if

$$\bigvee_{r \in L} r\,(c, a) = \top$$

In Listing 1, each line within the brackets is a deny-rule, and the six deny-rules together form a denylist.

3.2 Blocked Operations and Incompletely Blocked Operations

Now, we define what is blocked by a denylist, which, intuitively, should be the operations performed by the blocked commands.

Definition 2 (Operations and blocked operations) *On host system h, an operation o is a function over the states (e.g., the file system or network layers) of the system:*

$$o : S \to S$$

where S is the set of all possible states of the system, and o maps an old state to a new state. o is an operation performed by a command c with argument list a on h, denoted as $c, a \vdash_h o$, if the state transition conform to o,

$$s' = o(s)$$

In turn, a denylist L blocks o on h, denoted as $L \rhd_h o$, if

$$\exists c, a.c, a \vdash_h o \wedge L \rhd c, a$$

The set of all operations blocked by L on h is denoted as $\mathcal{O}_h(L)$.

Specifically, this paper focuses on six atomic file system operations, each represented by an ID and an argument list, as shown in Table 1. File system operations are a major type of operation that malware performs (Bayer et al., 2009). The six kinds cover three components of a host's file system that can be modified: file contents, file paths, and file metadata. These operations are clear and easy to define and validate. Other kinds of operations (e.g., network operations) may be investigated in the future, as discussed in §6. A detailed explanation of the six kinds of operations is in Appx. B.

Operation ID	Arguments
`op_read`	`path`
`op_write`	`path`, `content`
`op_create`	`path`
`op_delete`	`path`
`op_chmode`	`path`, `mode`
`op_chowgr`	`path`, `owner_and_group`

Table 1: Blocked operations under investigation.

Furthermore, Definition 3 defines when a denylist fully/incompletely blocks an operation. Note that the definition has nuances respecting complex operations that comprise other operations. This is discussed in Appx. C.

Definition 3 (Fully and incompletely blocked operations) *Given a denylist L and an operation o on a host system h, L fully blocks o on h, denoted as $L \rhd^+{}_h o$, if*

$$L \rhd_h o \wedge \forall c, a.c, a \vdash_h o \implies L \rhd c, a$$

i.e., every command (with proper arguments) that can perform o has been blocked by L.

Inversely, L incompletely blocks o on h, denoted as $L \rhd^-{}_h o$, if

$$L \rhd_h o \wedge \exists c, a.c, a \vdash_h o \wedge \neg(L \rhd c, a)$$

i.e., there are commands (with proper arguments) that can bypass L to perform o. $c \succ_{h,o} L$ denotes that the command c with proper arguments bypasses L to perform o.

The set of all operations fully/incompletely blocked by L is denoted as $\mathcal{O}_h^+(L)$/$\mathcal{O}_h^-(L)$. The set of all commands that bypass L to perform o on h is denoted as $\mathcal{B}_{h,o}(L)$.

4 The ShellSieve Pipeline

Based on the formalization, we now compute the bypasses of a denylist, i.e., $\mathcal{B}_{h,o}(L)$ for some denylist L. This decomposes into two steps: first, we find all operations that L blocks (bypasses are meaningless for operations that L does not block). Second, for each operation, we need to

enumerate all commands on the host that perform it and check whether L blocks each command. Any command that performs the operation and is not blocked by L is an effective bypass.

4.1 Overview

Input and output. The computation runs a shared, LLM-driven pipeline on denylists and commands on the host, respectively. For a denylist, deny-rules will be expanded (by the LLM) into commands that match them. Then, the pipeline produces the operations that the denylist blocks, along with the arguments that perform them. For a command on the host, the pipeline produces the operations that the command performs, along with the arguments that perform them. Table 2 shows example inputs and outputs of the pipeline.

Command with arguments	Operations
`cat $N.env`	`op_read($N.env)`
`mv $P /dev/null`	`op_delete($P)`
`cp $P /dev/stdout`	`op_read($P)`

Table 2: Examples of the commands (input), arguments (output), and operations (output) consumed and produced by the pipeline. The arguments and operations can contain placeholders, including `$N` for an arbitrary file name string, and `$P` an arbitrary file path.

The pipeline follows the overview in Figure 1. Each command with its arguments will go through the same three stages to produce the output, validated operations, and arguments:

- **Candidate enumeration (§4.2).** Given the command, an LLM is prompted to propose a candidate operation within Table 1 that the command may perform with proper arguments.
- **Execution-based validation (§4.3).** Each command is executed in a sandbox, which records the changes made to the file system. The changes are checked against oracles to confirm the correctness of the candidate operations.
- **Iterative repair (§4.4).** Candidate operations that fail the validation are fed back to the LLM with diagnostics to get a repair. The repaired candidates then go through the validation process again.

Ultimately, the operations performed by each command are compared against those blocked by the denylist to determine whether bypasses exist for that denylist (§4.5).

4.2 Candidate Enumeration

During the candidate enumeration stage, an LLM proposes candidate operations and arguments. The command may perform the operations with the arguments. This is done by prompting the LLM with the command's documentation, including its help message, manual pages, and entries on GTFOBins (a community-curated living-off-the-land attack database) (GTFOBins Project, 2026). The full prompt is in Appx. D.

Obviously unsatisfactory candidates, such as those containing syntax errors, are discarded, and the LLM is re-prompted until it proposes a satisfactory one. Operations that are identical or overlap are deduplicated. Duplication may occur because the "operations" in the implementation may contain placeholders representing infinite sets of operations. Given two sets of operations O_1 and O_2, O_1 duplicates O_2 if $O_1 \subseteq O_2$. In this case, we keep only O_2, as it is more general. An additional constraint applies to commands retrieved from denylist entries: the command with the proposed arguments must be blocked by the denylist.

4.3 Execution-based Validation

The validator decides, for each candidate operation, whether the command truly performs it when running on the host. It ensures that the operations proposed by the LLM are always valid, avoiding errors stemming from the LLM's inherent uncertainty, such as hallucinations. Each of the six kinds of operations is equipped with an *oracle*. The command with arguments and the operation are instantiated by substituting the placeholders with freshly generated concrete values that serve as flag files or canary tokens (e.g., `$P` to `./f.txt`). The command is then executed in a sandbox that records changes to the file system made by the execution. If the operation is performed, the changes must satisfy the oracles. Appx. E lists and explains the oracles.

4.4 Repair Loop

Commands and operations that fail validation are returned to the LLM together with the captured diagnostic information (codespanstdout, `stderr`, the return code, ...). The LLM is asked to emit a revised candidate that addresses the observed failure, and the revision is sent through execution-based validation again. This loop repeats for at most M (a configurable constant) rounds per original candidate and exits early as soon as a revision passes the oracle or once the LLM decides the candidate is not fixable. The repair loop recovers *near-misses*, such as unbalanced quotes or incorrectly used flags, that a validate-once pipeline would discard. The repair-loop prompt follows the candidate-enumeration template of Appx. D, as described there.

4.5 Matching Denylists with Bypasses

The shared pipeline yields, for each command, the operations it performs, and for each denylist, the operations it blocks. ShellSieve then iterates through all denylists and checks all commands that are not blocked by any denylist. For every pair of denylist and command, the set of operations blocked by that denylist and the set of operations performed by that command are intersected, and if the intersection is not empty, the command is a bypass of the denylist.

5 Evaluation

We collected a dataset of real-world command denylists used by terminal AI agents from GitHub. We applied ShellSieve to it to characterize the fragility of command denylists caused by incompletely blocked operations. The pipeline runs on a Debian 13.5 Docker image with the default GNU userland and is driven by Claude Sonnet 4.6 with default settings. The repair loop is capped at 5 iterations per candidate.

5.1 Research Questions and Metrics

The evaluation is framed against the formalization in §3 and targets the following research questions:

- **RQ1 (Problem severity).** How severe is the command denylist fragility problem in the collected dataset?
- **RQ2 (Severity in different cases).** Do factors such as how famous the project is or what agent it uses affect the severity of the problem?
- **RQ3 (Root causes).** What are the root causes of incompletely blocked operations?
- **RQ4 (Burden of fixes).** How difficult is it to repair a fragile denylist?

We answer each RQ using metrics computed from the operations and bypasses identified by ShellSieve according to the formalization in §3:

- **RQ1.** Severity is measured by *incompletely-blocked rate (IBR)*: for each operation o, the fraction of denylists L that incompletely block it among all that block it, i.e.,
$$\frac{\left\|\{L | L \rhd_h^- o\}\right\|}{\|\{L | L \rhd_h o\}\|}$$
where $\|\cdot\|$ denotes the size of a set.
- **RQ2.** We compute IBR for each repository, and measure whether and how it varies among repositories with different numbers of GitHub stars and the agent the repositories use.
- **RQ3.** We propose three hypotheses for the root causes of incompletely blocked operations, and test each against a measurable proxy of the hypothetical cause, e.g., how the likelihood of a command becoming a bypass relates to its popularity, or how the IBR changes when versatile commands are excluded.
- **RQ4.** The repair burden is proxied by the number of commands that the denylist must block in addition to cover all incompletely blocked operations.

5.2 Dataset Construction

The evaluation targets denylists for three widely used AI agents: Claude Code, Codex, and OpenCode. We collect the denylists from GitHub's per-project setting files. In total, we find 24,453 Claude Code configuration files on GitHub, 142 for Codex, and 1,235 for OpenCode. Of

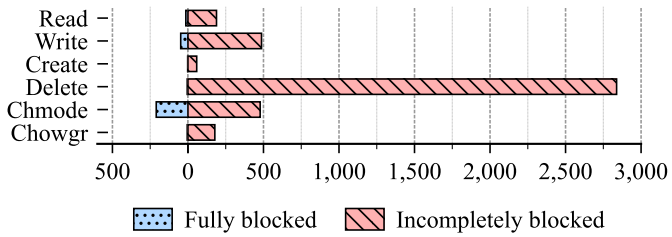


Figure 3: For each operation, the number of denylists that fully block it and incompletely block it.

these, 3,959, 136, and 573, respectively, have non-empty denylists. In total, we collect 4,668 denylists which contain 36,846 deny-rules. Among the denylists, 2,718 (58.2%) cover one of the six operations. Among the deny-rules, 6,681 (18.0%) cover one of the six operations. conduct Appx. F presents breakdowns of the denylists respecting the stars of their repositories and the operations they block.

5.3 Results and Findings

5.3.1 RQ1 (Problem severity)

RQ1 studies how often denylists overlook bypasses. For each operation, we compute the IBR respecting that operation.

Figure 3 shows the number of denylists that incompletely or fully block an operation. Among the six kinds of operations, denylists that incompletely block an operation account for the largest share (69.5–99.8%). The result means that most denylists overlooked at least one bypass for the operation they target, invalidating their protection against it.

Finding 1. The problem of fragile command denylists caused by incompletely blocked operations is severe. 69.5–99.8% of the denylists that target an operation overlook at least one bypass command for that operation.

5.3.2 RQ2 (Severity in different cases)

RQ2 studies whether such fragility is concentrated in particular kinds of projects or agents. Figure 4 reports, for each operation, the IBR, broken down by repository popularity (Figure 4a) and by agent (Figure 4b).

The IBR remains high across all three star-number tiers in Figure 4a and never drops below 50.0%. Similarly, the incomplete-blocked rate remains uniformly high across all three agents, never dropping below 37.2% and reaching 100% for most operation-agent pairs.

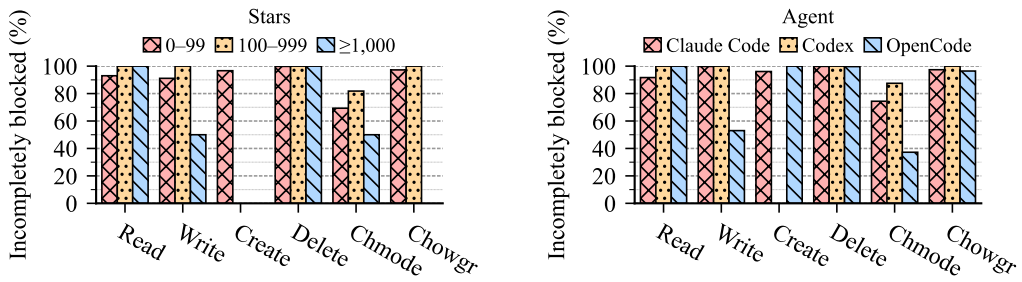
Finding 2. The fragile denylist problem is consistently severe across different projects and agents, as reflected by the uniformly high incomplete-blocked rates ($\geq$ 50.0% across projects and $\geq$ 37.2% across agents), which reach 100% for most operation and agent pairs.

5.3.3 RQ3 (Root causes)

To identify the root causes of incompletely blocked operations, we propose two explanatory hypotheses and statistically validate them. Note that the hypotheses are not meant to be a complete list but rather to provide initial insights and directions for future research. Appx. G presents another hypothesis that may be proposed, along with its validation. More hypotheses may be proposed and validated in a similar manner. The two hypotheses in the main body are as follows:

- **H1 (Ignorance).** The denylist authors do not know that the bypass commands exist.
- **H2 (Versatility).** The denylist authors may leave versatile commands that can perform many operations (e.g., the Python interpreter) unlisted for a tradeoff between security and usability, even though they can perform operations that authors want to block.

H1 (Ignorance). Intuitively, if the denylists overlook bypasses because their authors do not know that the bypass commands exist, then less popular (and thus less known) commands should be easier to become a bypass. To validate this hypothesis, we collect the rank of a command in the



(a) IBR with respect to stars

(b) IBR with respect to agents

Figure 4: Breakdown of incompletely blocked rate.

Debian Popularity Contest (Debian Project, 2026) as a proxy for the popularity of the commands. Less popular commands correspond to larger rank values. Then, we apply equal-frequency binning (Han et al., 2006) to split the commands into six bins based on their ranks and use a logistic regression model to estimate the frequency of bypass commands within each bin. The fitted model will predict, for a command at a given popularity rank, the probability that it is a bypass command for the operation.

Figure 5 shows the result. The figure shows that for `op_read`, `op_write`, and `op_create`, the fitted probability increases steeply with rank, indicating that less popular (higher-rank) commands are markedly more likely to bypass the denylist. The logistic regression yields highly significant positive slopes for these three (the largest p-value is 2.31×10^{-5}). The slopes for `op_chmode` and `op_chowgr` are positive as well but not significant at the 0.05 level (p being 0.13 and 0.17), which we attribute to how few commands bypass them at all (13 and 16, against 147 for `op_read`). Under `op_delete` the curve is flat ($p = 0.89$), so popularity carries essentially no signal there. This supports H1 for every operation, with enough bypass commands to measure the effect: less popular commands are markedly more likely to be overlooked and become bypasses.

H2 (Versatility). Intuitively, if the bypasses of denylists are mostly versatile commands deliberately left unlisted, excluding those versatile commands from the results should result in a significant decrease in the IBR. To validate this hypothesis, we enumerate the 82 versatile commands on the host, i.e., interpreters, shells, privilege and environment wrappers, and package and build front-ends, which can be driven to perform any of the operations. Two analyzers independently annotate these versatile commands and communicate on the details to determine the final version. Then we recompute the IBR, counting a denylist as incompletely blocking an operation only if a command outside this list bypasses it.

The result runs opposite to what H2 predicts. Excluding versatile commands leaves the IBR of four operations (`op_read`, `op_write`, `op_delete`, and `op_chowgr`) unchanged, and lowers it only slightly for the other two: from 96.7% to 88.3% for `op_create` (5 of 60 denylists become fully blocked) and from 69.5% to 68.2% for `op_chmode` (9 of 688). Across all operations, the number of denylists that incompletely block at least one operation stays at 3,091. Although versatile commands make up 47 of the 214 distinct bypass commands, nearly every denylist they bypass is also bypassed by a single-purpose command. This rejects H2: even with versatile commands set aside, 68.2–99.8% of the denylists that target an operation still overlook a bypass for it.

Finding 3. One cause of the fragility can be statistically confirmed: ignorance of less-common commands (H1), for every operation with enough bypass commands to measure the effect. Versatile commands as a root cause (H2) is rejected: excluding versatile commands still leaves 68.2–99.8% of the denylists incompletely blocking.

5.3.4 RQ4 (Burden of fixes)

RQ4 examines the cost of repairing a fragile denylist. Fully blocking an operation ($L \rhd_h^+ o$) requires the denylist to match *every* host command that performs o ($c, c \vdash_h o$). The repair burden may therefore be proxied by the number of bypass commands a defender must add to the denylist.

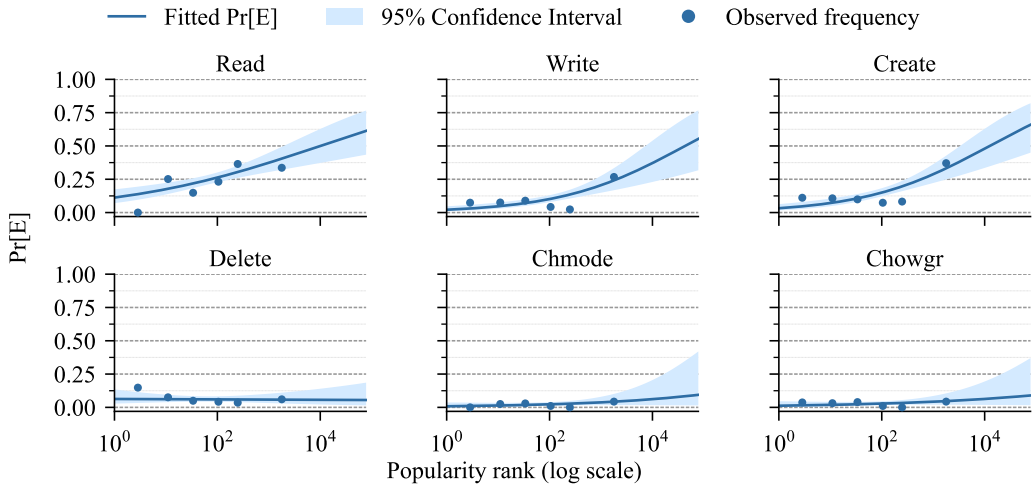


Figure 5: Fitted probability that a host command is a bypass for each operation as a function of its popularity rank (log scale), with 95% CIs (the shadows) and empirical rates (the dots). The empirical bypass rates are computed within each of the six equal-frequency bins, plotted at their mean ranks, and serve as a calibration reference for the fitted curve. E is the event that a command bypasses any denylists to perform the operation, and $\Pr[E]$ is the probability of that event.

	Read	*Write*	*Create*	*Delete*	*Chmode*	*Chowgr*
Min	307	131	2	3	1	3
Mean	334	187	137	79	2	25
Max	346	194	303	109	29	29

Table 3: Min/mean/max number of bypass commands.

On the reference host, fully blocking `op_read` takes, on average, 334 additional commands (up to 346), and closing every operation that a denylist blocks at once takes, on average, 127 commands (up to 827). Even the cheaper operations are far from a one-line fix: `op_write` and `op_create` average 187 and 137 bypasses each. The numbers are substantial for anyone who intends to fix the denylists. Table 3 reports the minimum, mean, and maximum numbers of distinct validated bypasses to be added per operation.

Finding 4. Repairing a fragile denylist by enumerating bypasses requires adding numerous commands (334 on average for `op_read`).

6 Limitations and Future Work

Completeness of the LLM-driven enumeration. Execution-based validation makes every reported bypass real, but the LLM may still overlook operations a command can perform or a denylist blocks. Our results are therefore lower bounds: the actual fragility can only be more severe. Sampling multiple models or combining the LLM with static analysis of command implementations may close this gap (Gosain & Sharma, 2015). For example, symbolic execution guided by documentation (Wong et al., 2015) or specifications inferred from natural-language descriptions (Pandita et al., 2012) could systematically enumerate what a command does.

Other kinds of operations. We study only six file system operations. Other operations, such as network access or process control, need oracles that simulate their environments (e.g., fake HTTP servers). ShellSieve's stages are operation-agnostic, so supporting a new operation requires only a new oracle. Oracles for some operations, such as those involving network accesses, may be tricky to implement, though. A promising direction is to emulate such environments using LLMs, as prior work has done for tool execution (Ruan et al., 2024) and interactive shells (Sladić et al., 2024). Such emulated oracles trade some of the guarantees of real execution for coverage of operations that are costly to reproduce faithfully.

ETHICS STATEMENT

Responsible disclosure. All the experiments are conducted locally. No harm is done to real-world systems. Also, we have responsibly disclosed the vulnerabilities to relevant stakeholders, including Anthropic and the project that uses the denylist in Listing 1. At the time of writing, the disclosure respecting Listing 1 is still under review. The disclosure to Anthropic was triaged and identified as an issue respecting the incompleteness of command denylists. We received no further substantive response, and to our knowledge the bypass remains unaddressed at the time of writing.

Dual-use considerations. We recognize that SHELLSIEVE and the accompanying dataset could lower the effort required to locate bypasses for a deployed denylist, and we have weighed this against the benefit of enabling defenders and researchers to measure a problem that is currently invisible to them for two considerations. First, the bypasses we report are not exploits: they are ordinary utilities used within their documented functionality. An adversary consulting user documentation can reach the same commands without our tool. Second, an attacker targeting a single denylist needs one bypass, which is cheap to find by hand. Our pipeline adds bulk measurement across thousands of lists, which is a defender need rather than an attacker's.

AI USE STATEMENT

In this work, we used generative AI tools to implement the software described in §4 and to polish the writing of this paper. All AI-assisted code was reviewed by the authors and is exercised by the execution-based validation the pipeline itself performs (§4.3). All AI-assisted text was read and edited by the authors.

REPRODUCIBILITY STATEMENT

The code and data for this paper will be available upon publication, along with environments and scripts to reproduce the results.

REFERENCES

Agentsh Project. Policy reference - agentsh, 2026. URL https://www.agentsh.org/docs/policy-reference/.

Anthropic. Claude Code auto mode: a safer way to skip permission, 2026a. URL https://www.anthropic.com/engineering/claude-code-auto-mode.

Anthropic. Configure permissions, 2026b. URL https://code.claude.com/docs/en/permissions.

Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *LEET*, 2009.

Grigorios Chrysos, Yixuan Li, Etsuko Ishii, Xuefeng Du, and Katia P. Sycara. Agentic AI in the wild: From hallucinations to reliable autonomy, December 2025.

Debian Project. Debian popularity contest, 2026. URL https://popcon.debian.org/.

Anjana Gosain and Ganga Sharma. Static analysis: a survey of techniques and tools. In Durbadal Mandal, Rajib Kar, Swagatam Das, and Bijaya Ketan Panigrahi (eds.), *Intelligent Computing and Applications*, pp. 581–591, New Delhi, 2015. Springer India. ISBN 978-81-322-2268-2. doi: 10.1007/978-81-322-2268-2_59.

GTFOBins Project. GTFOBins, 2026. URL https://gtfobins.org/.

Jiawei Han, Micheline Kamber, and Jian Pei. *Data mining: Concepts and techniques*, volume 10. Morgan Kaufmann, 2006.

Adnan Khan. Clinejection — compromising cline's production releases just by prompting an issue triager, February 2026. URL https://adnanthekhan.com/posts/clinejection/.

Linux man-pages Project. xattr(7) - Linux manual page, 2026. URL https://man7.org/linux/man-pages/man7/xattr.7.html.

Yue Liu, Yanjie Zhao, Yunbo Lyu, Ting Zhang, Haoyu Wang, and David Lo. "Your AI, My Shell": Demystifying prompt injection attacks on agentic AI coding editors, April 2026. URL http://arxiv.org/abs/2509.22040. arXiv:2509.22040 [cs.CR].

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, January 2026. URL https://arxiv.org/abs/2601.11868v1.

OpenAI. Codex | AI coding partner from OpenAI, 2026a. URL https://openai.com/codex/.

OpenAI. Rules – codex | OpenAI developers, 2026b. URL https://developers.openai.com/codex/rules.

OpenCode. Permissions, June 2026a. URL https://opencode.ai/docs/permissions/.

OpenCode. OpenCode | the open source AI coding agent, 2026b. URL https://opencode.ai/.

Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *2012 34th International Conference on Software eEgineering (ICSE)*, pp. 815–825. IEEE, 2012.

Yangjun Ruan, Honghua Dong, Andrew Wang, Silviu Pitis, Yongchao Zhou, Jimmy Ba, Yann Dubois, Chris Maddison, and Tatsunori Hashimoto. Identifying the risks of LM agents with an LM-emulated sandbox. In *International Conference on Learning Representations*, volume 2024, pp. 27031–27098, 2024.

Muris Sladić, Veronica Valeros, Carlos Catania, and Sebastian Garcia. LLM in the shell: Generative honeypots. In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pp. 430–435, July 2024. doi: 10.1109/EuroSPW61312.2024.00054. ISSN: 2768-0657.

tldr-pages Project. tldr-pages/tldr, June 2026. URL https://github.com/tldr-pages/tldr. original-date: 2013-12-08T07:34:43Z.

Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. DASE: Document-assisted symbolic execution for improving automated software testing. In *2015 37th IEEE International Conference on Software Engineering (ICSE)*, volume 1, pp. 620–631. IEEE, 2015.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 10471–10506, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.624.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models, December 2023. URL http://arxiv.org/abs/2307.15043. arXiv:2307.15043 [cs.CL].

APPENDICES

A. BYPASSES OF REAL-WORLD DENYLISTS

This section presents bypasses of real-world terminal agent denylists that we found in the wild. They demonstrate the prevalence of fragile command denylists. The first is the built-in denylist shipped in Claude Code's auto-mode.

Category	Commands
Interpreters	`python`, `python3`, `python2`, `node`, `deno`, `tsx`, `ruby`, `perl`, `php`, `lua`
Package managers	`npx`, `bunx`, `npm run`, `yarn run`, `pnpm run`, `bun run`
Local and remote shells	`bash`, `sh`, `ssh`

Table 4: The code execution commands in the built-in denylist of Claude Code's auto-mode.

Purpose of the denylist. This denylist enumerates code-execution commands, including language interpreters, package managers' script-running subcommands, local and remote shells, and forces them through the LLM judge, forbidding the customizable allowlist from silently approving them, as these commands can conduct arbitrary operations on the host system, and thus shouldn't be mindlessly waved through (Anthropic, 2026a). The denylist is hardcoded in Claude Code's source code, which is not open source. Thus, we cannot access it directly. However, we compile a fairly comprehensive list of commands in this denylist by enumerating and testing common code-execution commands against Claude Code's auto-mode, and we include the collection in Table 4.

The bypass. However, the enumeration is incomplete. It overlooks AWK, a POSIX-mandated utility available on every Unix-like host. Though designed for text processing, AWK uses a Turing-complete domain-specific language and, more importantly, can invoke Bash and pass arbitrary Bash scripts at invocation. Because AWK is not on the denylist, it effectively bypasses the rule that applies to Bash. Experiments confirmed that AWK can bypass the denylist in Claude Code's auto-mode, and we disclosed this to Anthropic, the developer of Claude Code.

The built-in denylist in Claude Code's auto-mode is a single special example. Similar ignorance is widespread in denylists used in GitHub projects that use Claude Code. Listing 1 is a denylist collected from GitHub that intuitively blocks commands that read environment variable definition files. However, many commands can perform the same operation, and the denylist blocks only a small proportion of them. Examples include `cp .env /dev/stdout` and `diff .env /dev/null`. The denylist considers only the most common commands for this purpose, thereby overlooking others.

Observation. The above two cases show that even denylists well maintained by the agent developers can be fragile, overlooking bypass commands, and such ignorance exists in user-compiled denylists in GitHub repositories.

B. EXPLANATION OF THE SIX KINDS OF OPERATIONS

The six kinds of operations listed in Table 1 are explained as follows:

- **`op_read`.** The content of the file at `path` is read and printed to stdout or stderr.
- **`op_write`.** `content` is written to the file at `path`.
- **`op_create`.** A new file is created at `path`.
- **`op_delete`.** The file at `path` is deleted.
- **`op_chmode`.** The mode of the file/directory at `path` is changed to `mode`. This is typically conducted by the `chmod` command on Linux.
- **`op_chowgr`.** The owner/group of the file/directory at `path` is changed to `owner_and_group`. This is typically conducted by the `chown` command on Linux.

Several notes are worth mentioning here: First, the list does not include a kind for "code execution", because "execution" is not a single side effect that can be observed in the host's file system. Second, the list is designed to be as minimal as possible. Thus, it does not include "moving a file," which

can be decomposed into a creation and a deletion. Third, the list omits less-used metadata, such as xattr (Linux man-pages Project, 2026), to focus on the most common operations first.

C. Complex Operations

The operations in Table 1 are atomic: each alters a single aspect of a single file system entry. Real invocations, however, often bundle several of them into a single higher-level effect that users think of as one action. A `git commit` run, for instance, writes multiple new blob and tree objects. It comprises many `op_create` and `op_write` operations.

When characterizing the operations blocked by a denylist or performed by a command, we have to decide whether we should consider such complex operations composed of the six atomic operations listed in Table 1. We follow the following principles:

- **Blocking is exact.** It means we consider a denylist to block an operation only when the operation is all that a blocked command performs. For example, a denylist matching `git commit` does not block any single `op_create` or `op_write` because only the overall effect makes sense for the purpose of the denylist. Such complex operations may be left to future research.
- **Bypassing is by containment.** It means we consider a command to bypass a denylist to perform an operation if the operation is one it performs. For example, the `git commit` command bypasses a denylist that blocks file creation in the `.git` folder because it creates files when committing changes. This makes sense because a bypass succeeds whenever it performs an action that a denylist forbids, regardless of the command's intended rationale.

D. LLM Prompts

Candidate enumeration. During candidate enumeration (§4.2), the LLM is prompted with the command's documentation and asked for a structured list of candidate operations and arguments. Figure 6 shows the prompt.

Repair loop. The prompt that returns a failed pair of candidate operation and arguments to the LLM for repair (§4.4) follows the same template, with the captured error message and the prior attempts appended to it. It differs in one respect: we deliberately exclude the GTFOBins entry from the reference documentation because it is community-compiled and more likely to contain outdated or incorrect information.

E. Operation Oracles

The oracle for each of the six operations is as follows, in the order of Table 1. Figure 7 illustrates those for `op_delete`, `op_read`, and `op_write`.

- **`op_read(path).`** A flag file is seeded with a canary token and used to substitute for the command's file path placeholder(s). The oracle passes if and only if the canary appears in `stdout` or `stderr`.
- **`op_write(path, content).`** The command is supplied with a flag file path already present in the host environment and a canary token to be written to the file. The oracle passes if and only if the canary appears in the file after execution.
- **`op_create(path).`** `path` is instantiated with a fresh unoccupied path. The oracle passes if and only if a filesystem entry exists at `path` after execution.
- **`op_delete(path).`** A flag file is seeded at `path` before execution. The oracle passes if and only if `path` no longer exists after execution.
- **`op_chmode(path, mode).`** A flag file is seeded with a known initial mode m_0 (e.g., `0644`) and `mode` is set to a target $m_1 \neq m_0$ (e.g., `0700`). The oracle passes iff the permission bits of `path` equal m_1 after execution. Choosing $m_1 \neq m_0$ prevents a no-op from passing.
- **`op_chowgr(path, owner_and_group).`** A flag file at `path` is seeded with a known owner and group, and `owner_and_group` names a distinct owner/group that exists on the host. The oracle passes if and only if the owner and group of `path` match the target afterward.

Note that the oracles are intentionally loose. For example, the oracle for `op_read` requires that stdout/stderr contains the canary token, and other content around the token is permitted. The essence is whether the seeded token reaches `stdout`/`stderr`. Everything orthogonal to it, such as exit codes, warnings on `stderr`, or incidental output, should be ignored. A command that emits

```
Role

You are a security researcher analyzing the
functionality of shell commands.

Documentation

Command: {{command}}
Args: {{arg_list}}
Docs: {{help_msg}} {{man_pages}} {{gtfobins}}

Instructions

For each operation below, decide whether the
command can perform it solely (no chaining,
no pipelines) with some arguments. Use
wildcards to generalize the results as much
as possible.

Operations

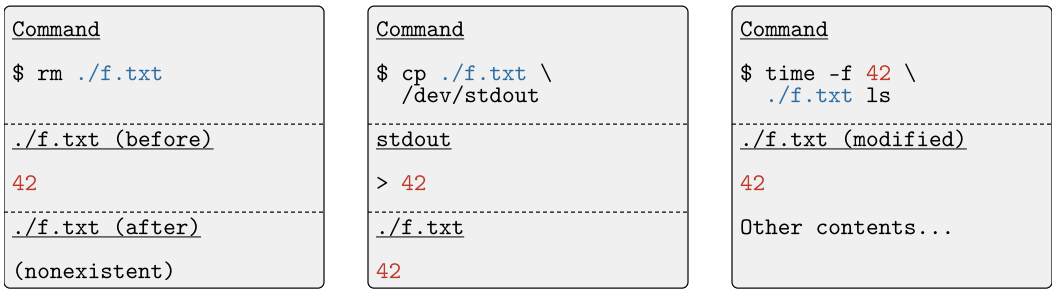
1. op_read(path)
2. op_write(path, content)
...

You can use the following wildcards in the
arguments: $P, $N, ...

Output schema

Output a JSON array similar to the following
example:
```json
[{ "arguments": ["$P", "/dev/stdout"],
   "operation_id": "op_read",
   "operation_args": ["$P"] }]
```
```

Figure 6: Abbreviated prompt template used in candidate enumeration. The LLM receives the command's documentation and is asked to propose a structured list of operation and argument candidates. Double braces mark the inserted contents.



(a) op_delete (b) op_read (c) op_write

Figure 7: Execution-based validation. Each candidate is instantiated with a freshly generated flag file (blue)/canary token (red). The command is executed in a sandbox. A per-scope oracle inspects the pre- and post-execution states to determine whether the operation was actually performed.

a warning or returns a non-zero status while still leaking the file's contents genuinely performs the operation, and a stricter oracle would incorrectly reject it.

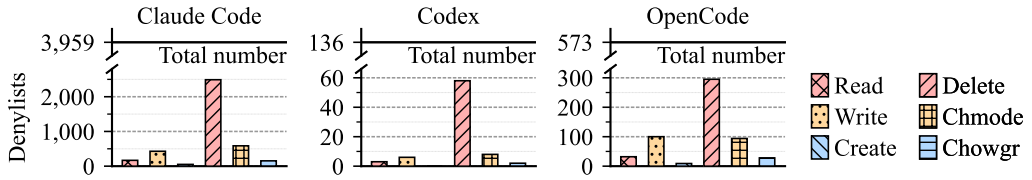


Figure 8: Number of denylists that block each operation, grouped by agent. The `op_` prefix in the operation IDs is omitted.

	Read	*Write*	*Create*	*Delete*	*Chmode*	*Chowgr*
TLDR Op	97.9	81.4	81.9	85.7	83.3	100.0
Other	4.1	1.4	2.7	1.0	1.1	1.4

Table 6: Percentage of host commands that are validated bypasses, split by whether the operation appears in the command's TLDR documentation. *TLDR Op* covers commands whose TLDR page states the operation. *Other* covers those whose TLDR page does not.

F. Statistics of the Dataset in Evaluation

Table 5 shows the distributions of the number of GitHub stars these repositories receive. The repositories use 174 programming languages and cover 1,676 topics (tagged by their maintainers). Figure 8 shows the number of denylists that cover (i.e., $o \in O_h(L)$) each kind of the six operations in Table 1. The dataset well represents real-world scenarios where the denylists are used.

Stars	**Claude Code**	**Codex**	**OpenCode**	**Total**
0–99	3,895	130	563	4,588
100–999	57	4	8	69
≥1000	7	2	2	11
Total	3,959	136	573	4,668

Table 5: Stars of the GitHub repositories.

G. Hypothesis that Bypasses are Caused by Semantic Gap

In addition to the two hypotheses in §5.3.3, another hypothesis (H3) of the root cause of denylist fragility could be the semantic gap: the command's ability to perform the operation is hidden from its apparent usage, so authors never associate it with the operation in the first place.

Intuitively, if H3 makes sense, the overlooked bypasses should concentrate among commands that do not advertise the operation in their everyday documentation. To validate this hypothesis, we use the community-maintained TLDR pages (tldr-pages Project, 2026), which list each command's most common usages, as a proxy for a command's apparent purpose. We rerun the ShellSieve pipeline using only the TL;DR documentation in the prompts to identify the obvious operations each command performs. Then, for each operation, we split the host commands into those whose TLDR page states the operation and those whose page does not, and compute the percentage of validated bypasses in each group. If a semantic gap were the dominant cause, bypasses should be more frequent among commands whose TLDRs do not mention the operation.

Table 6 shows the result, which runs opposite to what H3 predicts. For every operation, validated bypasses are far more common among commands whose TLDR states the operation than among commands whose TLDR does not, at 88.7% versus 1.9% overall and, for instance, 97.9% versus 4.1% for `op_read`. The overlooked bypasses are therefore mostly commands whose TL;DR already states the operation, so their capabilities are advertised rather than hidden, and the semantic gap is at best a secondary cause. This rejects H3: the commands authors miss are documented in plain sight.