NATURAL CODE TRANSFORMATION CONFUSES AGENTS IN VULNERABILITY ANALYSIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Having demonstrated a strong potential for automating vulnerability discovery, AI agents can also be used to attack software systems at scale. Today's software systems need mechanisms to level the playing field by slowing down potential attacks, giving them time to patch vulnerabilities and strengthen software. Along this line, our study of agent trajectories in vulnerability analysis shows a surprising finding: agents produce dramatically different results across real-world code commits that carry the same functionality and the same vulnerability. This phenomenon appears consistently in multiple code domains and vulnerability categories.

It motivates us to study *natural* and *functionality-preserving* code transformations derived from real-world software development, and their potential as a defense against attacker agents. We build a corpus of natural transformations by mining recurring development patterns from real-world code commits, and apply them to evaluate impact on agentic vulnerability analysis. Our results show that even a single form of transformation can effectively "confuse" frontier agents, e.g., Codex with GPT-6 Sol, reducing successes in vulnerability discovery. We also study different agent failures, whether transformations can be composed to provide stronger protection, and adaptive attackers who seek to reverse them. Overall, our study of natural code transformation significantly expands the research space of using code transformations to defend against malicious agents.

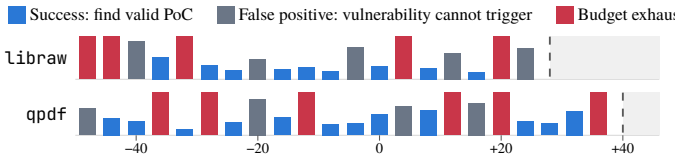
1 INTRODUCTION

Frontier AI agents can now analyze large volumes of software code to identify vulnerabilities (Wang et al., 2026b; Lee et al., 2026). Bad actors can use them as powerful tools to discover and subsequently exploit zero-day vulnerabilities in widely used software (Carlini et al., 2026; Wang et al., 2026a; Lee & Brumley, 2026). Yet many companies and institutions lack the computational resources to scan their own software using advanced agents, leaving them vulnerable to attackers wielding more powerful ones. Less resource-rich stakeholders need a way to level the playing field, a way to slow down attackers long enough to give themselves time to identify and repair their own software.

In our analysis of agent trajectories in CyberGym (Wang et al., 2026b), we make surprising observations that different versions of the same codebase often exhibit varying levels of resilience to AI agents scanning for vulnerabilities (Section 2.2). More specifically, within a single repository, the agent's success rate in vulnerability analysis fluctuates significantly across adjacent code commits that carry the same functionality and the same vulnerability (Figure 1). In many cases, the agent either fails to identify the vulnerability after exhausting the budget or generates false positives in which the proof-of-concept (PoC) input cannot trigger the vulnerability. Such a pattern is consistently observed across multiple code domains and vulnerability categories.

This finding suggests that agents' ability to identify software vulnerabilities is highly dependent on the code representation (i.e., syntax and structure) used in real-world software development. It further prompts us to consider the concept of *natural and functionality-preserving code transformations*, and their potential as a defense to slow down or even stall attacker agents. Conceptually, natural, functionality-preserving code transformations are like "imperceptible adversarial perturbations" in the image domain (Szegedy et al., 2013), which are intentional content modifications that preserve the original content's primary utility while manipulating how it gets interpreted and used by AI models.

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.



(a) Agent results on earlier and later commits of the same project (% of token budget)

Project	Agent success	False positive	Budget exhausted	Result flip
libraw	**9**/19	**5**/19	**5**/19	**12**/18
qpdf	**12**/22	**4**/22	**6**/22	**14**/21

(b) Detailed results breakdown

Figure 1: Vulnerability analysis results of frontier agents (GPT-5.6 Sol with Codex in this example) vary significantly across commits in the same software. Commit 0 is the original version evaluated by the agent; +/- denotes preceding/following commits; the dotted line marks the vulnerability-fixing commit. Bar color = agent results; bar height = agent token cost (% of budget). The table summarizes result variance, with "flip" indicating the agent's result changed between consecutive commits.

Prior work has studied fixed code obfuscations for testing code models' robustness, applying a predefined set of local syntax-preserving code rewrites, e.g., variable renaming, equivalent control-flow rewrites like `while`-to-`for`, and unreachable-code injection (Wang et al., 2023; Mahmud et al., 2026). However, these transformations are limited in both diversity and impact on agents (Section 4.1); some of them are easily identifiable and thus trivially reversible (Li et al., 2025b).

In contrast, we recognize that real-world software development activities, e.g., those exposed in commit histories, provide opportunities to identify natural code transformations that capture a much broader range of structural and semantic code changes, enabling diverse instantiations across code. Since these transformations occur naturally during normal software development, they leave few discernible clues for agents to recognize and reverse. These considerations make them attractive as potential techniques for code hardening against attacker agents.

To explore the feasibility of this approach, we mine recurring patterns from a large volume of real-world code commits (Muennighoff et al., 2024), distill them into reusable transformation skills (Li et al., 2026; Jiang et al., 2026), and instantiate and compose them into diverse implementations based on the code context. This leads to a large space of transformations resilient to adaptive agents attempting to reverse them. We validate transformed codebases to ensure that functionality and vulnerabilities are preserved modulo the ground-truth tests (Le et al., 2014), and evaluate their effectiveness in slowing down or stalling a variety of frontier agents. Our key findings include:

- *Natural transformations substantially reduce frontier agents' success:* A single form of natural transformation can already effectively "confuse" frontier agents and lower their success rates significantly. For GPT-5.6 Sol with Codex, the agent's mean success rate falls from 63.3% to 36.8% (−42%) under natural transformations, compared with 49.7% (−21%) under synthetic obfuscations. Among the agent failures, 80% result from the agent generating a wrong PoC that cannot trigger any vulnerability, while 20% are due to budget exhaustion.
- *Composition of transformations can be even more effective.* For GPT-5.6 Sol with Codex, combining two or three transformations reduces agent success by up to an additional 40% relative to the best single-transformation condition (Table 2). The agent's success rate even drops to 20.0%, compared with 63.3% without transformation.
- *Adaptive code rewriting by the agent remains ineffective.* Even when told which transformation was applied, the agent (GPT-5.6 Sol with Codex) recovers 25.9 percentage points of success on synthetic obfuscations on average, but only 3.5 points on natural transformations. Its success on our transformed repositories remains 30.1 points below that on the original code.
- *More computation does not close the success gap within the tested limits.* With 4× the default budget, the agent's success on transformed repositories still trails that on the original repositories by 19.8 points, and at max reasoning effort it still trails by 25.1 points. The gap varies by at most 2.0 points across 1 to 9 repeated runs per task.

Our work validates the feasibility of mining real-world coding activities to identify rich and diverse code transformations that can protect source code against AI agents scanning for vulnerabilities. In Section 5, we highlight several open challenges in translating this concept into a practical and robust toolkit for code hardening against agentic vulnerability analysis. Although related approaches have demonstrated effectiveness in traditional program analysis (van Tonder & Le Goues, 2020; Jung et al., 2019; Güler et al., 2019), extending them to agentic settings introduces new exciting open challenges.

2 SURPRISING OBSERVATIONS FROM REAL-WORLD CODE COMMITS

This work was motivated by our surprising observations from analyzing trajectories of AI agents that detect and reproduce software vulnerabilities in CyberGym (Wang et al., 2026b). We discuss these findings below, beginning with a brief discussion of the problem context.

2.1 RELATED WORKS AS CONTEXT

Agentic vulnerability analysis Driven by their ability to understand and reason about code, recent work demonstrates agents' strong potential in vulnerability detection by generating proof-of-concept (PoC) input (Wang et al., 2026b; Yildiz et al., 2025; Desai et al., 2026; Zhu et al., 2026b; Luo et al., 2026; Zhu et al., 2026a; Guo et al., 2025), patching (Wang et al., 2025; Shen et al., 2026; Zhang et al., 2026; Mündler et al., 2024), and penetration testing (Shen et al., 2025; Lin et al., 2026; Gioacchini et al., 2025; Shao et al., 2024). Agentic vulnerability analysis has also become the foundation for offensive tasks like exploit generation (Zhu et al., 2025; Wang et al., 2026a; Lee & Brumley, 2026; Zhu et al., 2026c; Carlini et al., 2026). Despite the strong reported performance, it remains unclear whether this performance is robust, e.g., stays consistent across variants of the same program, and can be potentially leveraged to harden the software against attackers. This gap motivates our work.

Indeed, prior studies show that code models can be sensitive to program representations (Mahmud et al., 2026; Honarvar et al., 2026; Thimmaiah et al., 2026; Li et al., 2025a; Hooda et al., 2025; Pei et al., 2023). This raises concerns that agentic vulnerability analysis benchmarks (Wang et al., 2026b; Lee & Brumley, 2026; Wang et al., 2026a; Lee et al., 2026) might overlook the impact of program representation and instead attribute performance differences solely to vulnerability characteristics and agent capabilities. Our work examines this concern by studying agent behavior under natural code transformations derived from real-world software development activities.

Robustness of code models Robustness of code models is typically evaluated using semantics-preserving perturbations (Yang et al., 2024; Hossen & Hei, 2024; Liu et al., 2025; Mahmud et al., 2026; Henkel et al., 2022; Applis et al., 2021; Gao et al., 2023) through manually crafted, context-agnostic code obfuscations, e.g., identifier renaming and dead-code insertion (Henkel et al., 2022; Mahmud et al., 2026; Li et al., 2025b). The local, syntactic nature of these transformations limits the diversity of program changes, often yielding recognizable, thus removable patterns (see Section 4.2). Our work targets a different goal and considers different types of transformations, i.e., those that naturally arise during software development and are widely observed in real-world codebases.

2.2 AGENTIC VULNERABILITY ANALYSIS IS SUSCEPTIBLE TO CODING ACTIVITIES

Real-world software development activities can change how code is organized and represented while leaving an existing vulnerability untouched. While vulnerability analysis typically targets a single version of the software (e.g., CyberGym uses a particular commit with a known vulnerability as the ground truth), real-world software often undergoes continuous maintenance represented by a sequence of code commits. The commits before and after the targeted version often differ only by minor changes, preserving the same functionality and often the same vulnerability. This motivates us to ask the question: *can an agent's success in identifying a vulnerability on a program transfer to neighboring commits sharing the same vulnerability?*

To answer this question, we consider projects from CyberGym in which the agent has successfully identified the ground-truth vulnerability, i.e., generated a PoC input. We then test the frontier code agent on the commits before and after this specific project version that also hold the same vulnerability (verified by replaying the PoC). For both the original program in CyberGym and the other versions in our sampled commits, the agent is explicitly tasked with detecting the ground-truth vulnerability.

Interestingly, we find that the agent's vulnerability analysis results vary significantly across commits, even though they have been extensively pre-trained by most frontier LLMs (Muennighoff et al., 2024). Figure 1a shows two projects included in CyberGym, while we have made consistent observations across diverse projects and LLM agents (Section 4.1). Its success rate in finding the ground-truth vulnerability fluctuates, often resulting in either failing to identify the vulnerability after exhausting the budget or generating false positives where the PoC cannot trigger the vulnerability.

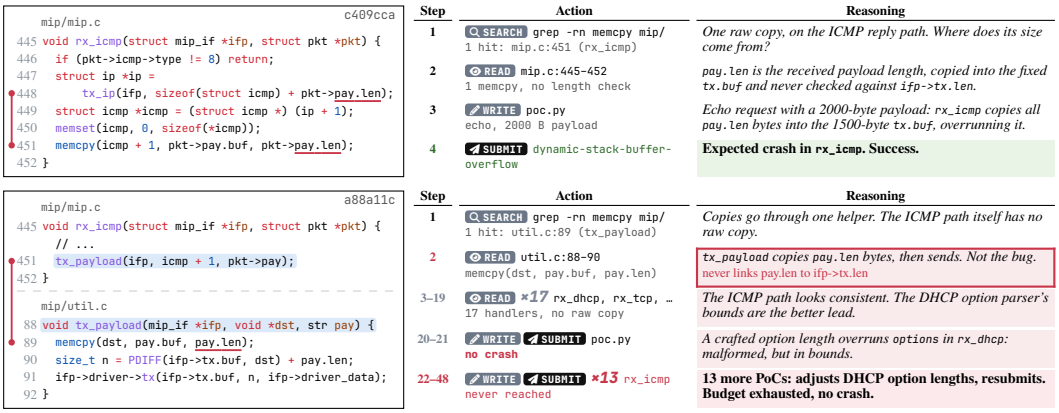


Figure 2: A small change between code commits, `c409cca` (top) and `a88a11c` (bottom), flips the agent's vulnerability analysis from success to failure. `a88a11c`'s simple code refactoring (extracting copy-and-transmit into `tx_payload`) sends the agent to a false lead, failing to reproduce the crash.

A concrete example Figure 2 illustrates how a small change in the program representation can redirect the agent's investigation. Two adjacent commits, `c409cca` (top) and `a88a11c` (bottom), in the `Mongoose` networking library (Cesanta, 2026) contain the same vulnerability and are functionally identical. The small code change is highlighted in `a88a11c`. The agent locates the vulnerability in `c409cca` (reflected by its "Action" and "Reasoning" results), but fails to locate any on `a88a11c`. The two commits differ only in that `a88a11c` places two function calls in separate files, but this separation makes it harder for the agent to link them, leading it to pursue false leads and exhaust the budget.

From real-world software evolution to natural code transformations Our findings above show that existing code development activities provide a rich set of *natural*, functionality-preserving code transformations that can elicit diverse agent behaviors and reasoning trajectories. Compared to existing work studying fixed obfuscations that manipulate simple code syntax (Section 2.1), real-world software evolution introduces a much broader (and unpredictable) range of structural and semantic code transformations. And since these transformations occur as part of normal code evolution, they are difficult to identify or to reverse. These properties make them attractive as potential techniques for a code hardening defense against attacker agents (Wang et al., 2026a; Lee & Brumley, 2026).

3 Mining and Applying Natural Code Transformations

Having established the potential benefits of natural code transformations, we now explore how to mine them from real-world development activities while keeping them broadly applicable to diverse codebases. First, we mine recurring development patterns, e.g., refactoring activities, from code commits and organize them into base transformations represented as agent skills. We then use a coding agent to localize applicable program points and implement these skills as concrete transformations. We evaluate their efficacy as a potential defense against vulnerability analysis agents in Section 4.

Mining transformation categories from existing codebases We group recurring development patterns into transformation categories, e.g., extracting duplicated cleanup code into a shared utility function. The categories abstract away commit-specific names and locations so the same transformation can be applied to other repositories. Specifically, we use CommitPackFT (Muennighoff et al., 2024), which contains 702,062 real-world commits across 14 major software development tasks (e.g., refactoring, cleanup, etc.) paired with instruction-like commit messages. Given that existing agentic vulnerability analysis targets primarily memory-unsafe languages, we focus on C and C++ commits. After removing duplicate commits and those with fewer than ten changed lines, we retain 1,196 C commits and 851 C++ commits for mining base transformations.

We iteratively apply DeepSeek-V4.1-Flash (DeepSeek-AI, 2026) to extract up to three atomic transformations per commit. Each action follows the form *action* + *concrete code element* + *purpose*, e.g., *extract into a helper* + *duplicated cleanup statements* + *to centralize resource release*. The transformation is described independently of the particular functions and variables modified by the commit.

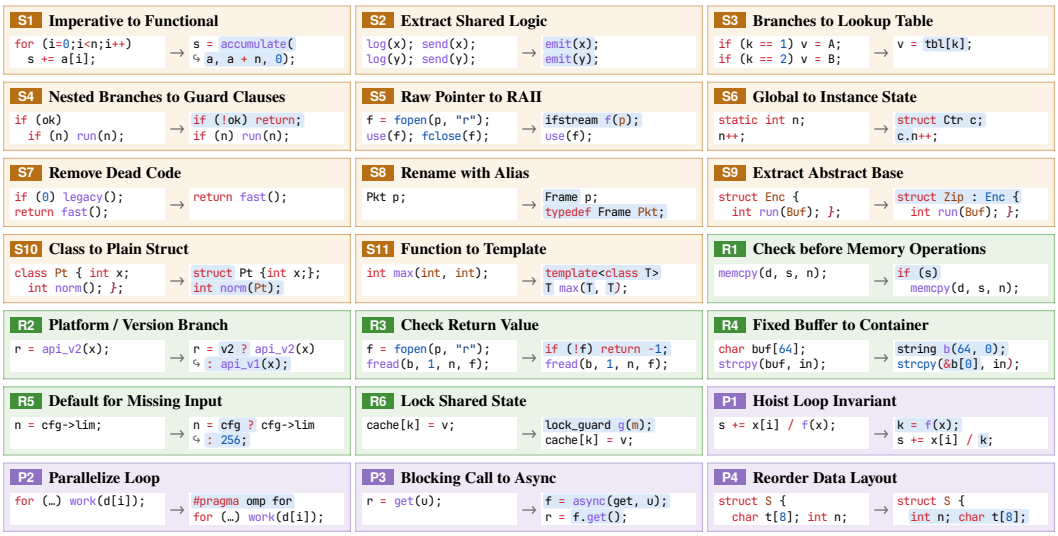


Figure 3: Top 21 most frequently occurring natural transformations identified by mining CommitPackFT, grouped into 3 categories: structure ( ), robustness ( ), and performance ( ). Each card shows a representative code snippet before and after the transformation. Each transformation is then represented as a skill, allowing diverse applications to target codebases. Unlike synthetic transformations, these transformations encode human intent and may be harder for agents to distinguish.

Instructions to guide the extraction include 15 manually written seed categories as in-context examples of the desired format and level of abstraction. Our instructions explicitly exclude identifier-only changes, added tests, and new features. We show them in our anonymous repository.

From transformations to skills We leverage `text-embedding-3-large` (OpenAI, 2024) to embed the transformation descriptions and apply Ward hierarchical agglomerative clustering (Ward Jr, 1963) to the L2-normalized embeddings. For each cluster, we inspect representative transformations and their supporting commits, remove superficial edits such as variable renaming, and merge overlapping categories at different levels of granularity. The resulting catalog contains 48 transformations: 25 for structure, 15 for robustness, and 8 for performance. For evaluation purposes, we select a subset of 21 transformations that occur the most frequently in our analysis, which already cover 84% of commits in the dataset. As shown in Figure 3, we represent each mined category as a skill that contains a concise instruction and a before–after code snippet (Li et al., 2026; Jiang et al., 2026). The instruction specifies the development action identified during mining, while the example illustrates the concrete implementation. For example, S2 instructs the agent to extract shared logic, with an example that replaces repeated statements with calls to a helper.

Such an abstract representation of transformations enables diverse, thus less predictable applications to target codebases. This significantly expands the search space faced by an adaptive attacker agent attempting to identify and reverse-engineer the transformations (Section 4.2), strengthening the robustness of the transformed codebase.

Instantiating transformations for a specific codebase We leverage DeepSeek-V4.1-Flash to apply the mined skills to the target repository by instantiating them as concrete transformations. Since agents have built-in stochasticity in their execution, applications of the same skill across instances can vary in the precise transformations and locations where they are applied, thus increasing the difficulty of detecting or reversing them. We also note that transformations can be composed simply by applying transformations sequentially in diverse orders. The specific ordering of composed transformations adds additional diversity, making reversing transformations even harder.

To ensure our transformed code matches the original in functionality, we rebuild/recompile the transformed variant and run the codebase's own test suites to ensure that the original and transformed versions are equivalent modulo the given tests (Le et al., 2014). We also replay the ground-truth PoC to ensure that the original vulnerability remains present. If these tests fail, we feed the errors back to the agent as feedback to iteratively reapply the transformations.

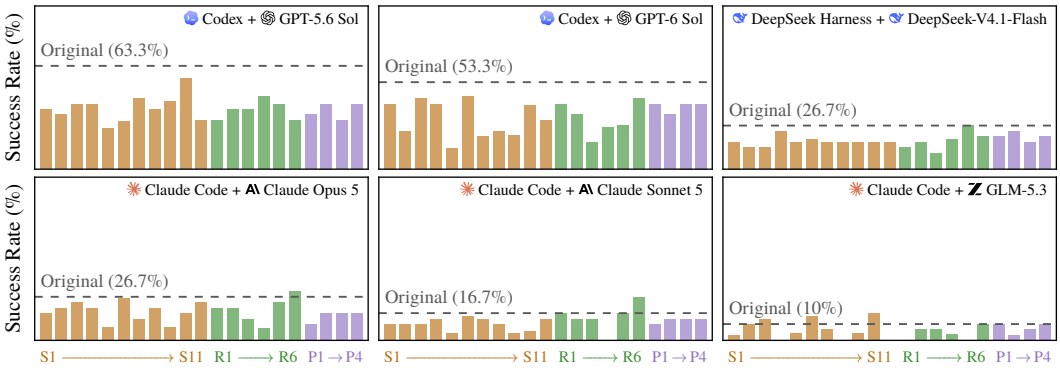


Figure 4: Agent's success rates (%) under each natural transformation described in Figure 3 for six model–harness combinations. Dashed lines mark the original success rates without transformations. All panels share the same scale. Detailed rates and relative changes are listed in Appendix B.

4 Evaluation

We now evaluate the impact of natural, functionality-preserving code transformations on agentic vulnerability analysis, i.e., whether/how they slow down or stall a variety of frontier agents.

Target code Our evaluation uses tasks from CyberGym (Wang et al., 2026b). With CyberGym's default budget setting ($2 or 100 steps per task), evaluating frontier agents and LLMs on all tasks across 21 transformations would require roughly $24K in API credits for one model–harness combination. We take the following steps to keep our costs manageable. First, since our study focuses primarily on robustness rather than performance, we only consider tasks solvable by existing agents with sufficient budget. Specifically, we consider tasks that GPT-5.6 Sol with Codex at extra-high reasoning successfully solves within 20 minutes. We further excluded tasks with trivial (zero-byte) PoCs and selected a subset of the remaining samples whose distribution across PoC size, crash depth, and other features was similar to that of the original dataset. This subset still results in approximately $10K in API credits due to the combinations of transformed variants, models, and agents, and the default three runs per transformed sample. CyberGym defines multiple task levels that expose different amounts of information about the target vulnerability to the agent. In our evaluation, we use level 0 as our primary setting, where agents receive the codebase and are tasked with generating PoC input that crashes the vulnerable version and passes the patched version.

Agent/LLM tested Given the reasoning-intensive nature of our evaluation, we focus primarily on models with strong reasoning capabilities and coding performance. We evaluate six state-of-the-art LLMs spanning both open-weights and proprietary models: DeepSeek-V4.1-Flash (DeepSeek-AI, 2026), GPT-5.6 Sol (OpenAI, 2026b), GPT-6 Sol (OpenAI, 2026c), Claude Opus 5 (Anthropic, 2026b), Claude Sonnet 5 (Anthropic, 2026c), and GLM-5.3 (GLM-5-Team, 2026) (we obtained permissions in accessing their cyber-capabilities, e.g., OpenAI Daybreak and Claude Cyber Verification Program). We pair these models with three widely used agent harnesses: Claude Code (Anthropic, 2026a), Codex (OpenAI, 2026a), and DeepSeek Harness (DeepSeek, 2026), based on known model-harness combinations on the CyberGym leaderboard (CyberGym, 2026). We exclude some top-ranking ones, e.g., Claude Mythos (Anthropic, 2026d), due to their limited public availability.

4.1 Impact of Code Transformations

Figure 4 shows the agent's success rate across all tasks for each transformation listed in Figure 3. Detailed success rate values, relative changes, and comparisons across transformations and model–harness combinations are provided in Appendix B. Across the LLM agents and transformations, the mean success rate falls from 32.8% on the original codebases to 20.4% across transformed ones, a mean relative drop of 36.2%. The largest relative drop occurs for Claude Sonnet 5 with Claude Code under R4, where the success rate drops all the way from 16.7% to 0.0%.

As the transformation agent is agnostic to the vulnerability (Section 3), a transformation is not tailored to obscure it and can occasionally make it easier to find. We observe such gains only in the bottom

three combinations of Figure 4, all with Claude Code. R6 raises the success rate of Claude Sonnet 5 from 16.7% to 26.7% and of Claude Opus 5 from 26.7% to 30.0%, and S10, S6, and S3 raise GLM-5.3 from 10.0% to 16.7%, 14.8%, and 13.3%, respectively. No transformation raises the success rate of the other three combinations. We investigate this further in Section 4.2 by reversing the transformations, e.g., for S3, `Branches to Lookup Table`, we study `Lookup Table to Branches`.

Comparison with synthetic transformations Table 1 compares our transformations mined from code development activities with six synthetic obfuscations extensively adopted in prior robustness evaluation of code models (Li et al., 2025b; Mahmud et al., 2026). We evaluate on GPT-5.6 Sol with Codex, since it has the highest original success rate. We find that on average, these transformations decrease the agent success rate from 63.3% to 49.7%, whereas our transformations decrease it to 36.8%. Two synthetic transformations are exceptions, i.e., identifier renaming and loop-to-recursive, as we observe they are highly effective in misleading LLMs, e.g., using confusing variable names (Yefet et al., 2020), but are easy to neutralize and make ineffective by adaptive attackers who also proactively transform the code, e.g., generate meaningful variable names, before the analysis (Section 4.2).

Table 1: Success rates (%) of GPT-5.6 Sol with Codex against synthetic transformations.

Transformation	%
Original	63.3
Ours (mean)	36.8
Remove comments	58.3
Reformat	66.7
Identifier renaming	23.3
`if-else` → `switch`	65.0
Insert deadcode	53.3
Loop to recursive	31.6

Composition Real-world code development often combines transformations of different kinds, so we compose transformations from different families by applying them in sequence and validating the ground-truth PoC after each step (Table 2). For GPT-5.6 Sol with Codex, we compose three widely applicable transformations in each group, i.e., S2, R3, and P1, into three pairs and one triple. Every composition degrades the agent more than its most effective constituent, by 3.3 to 13.3 points. The degradation compounds: the mean success rate falls from 34.4% with the three constituents alone to 26.7% with pairs and 20.0% with the triple. Composition can also make individually weak transformations effective. With Claude Code, R6 alone even increases the success rate of both Claude Opus 5 and Claude Sonnet 5 above the original, and S11 alone only lowers it by 3.3 points. However, when composed, R6 + S11 reduces the success rate of Claude Opus 5 from 26.7% to 16.7% and of Claude Sonnet 5 from 16.7% to 10.0%.

4.2 Sensitivity Analysis

We perform a series of ablations to determine whether code transformations consistently affect agent performance in various settings. Similar to Section 4.1, we use GPT-5.6 Sol with Codex as our primary model–harness combination. We select the two transformations most effective against this combination from each transformation family: structure (S5, S6), robustness (R1, R6), and performance (P3, P1).

Table 2: Success rates of composed transformations for GPT-5.6 Sol with Codex (top) and Claude Code (bottom). Individual success rates are shown in parentheses.

Transformation	Success rate (%)
S2 + R3	26.7 (33.3 / 36.7)
S2 + P1	23.3 (33.3 / 33.3)
R3 + P1	30.0 (36.7 / 33.3)
S2 + R3 + P1	20.0 (33.3 / 36.7 / 33.3)
(Opus 5) R6 + S11	16.7 (30.0 / 23.3)
(Sonnet 5) R6 + S11	10.0 (26.7 / 13.3)

Budget We evaluate whether changing the agent's budget impacts its success rate. We reran GPT-5.6 Sol with Codex on the original codebase and under the six transformations with per-task budgets of $1, $2, $4, and $8, with the results shown in Figure 5a. Success on the original codebase stops increasing at the default $2 budget with a success rate of 63.3%, after which it stays consistent, but the transformed repositories keep improving up to $4, plateauing around 41.7% with no further improvements from increasing the budget. While increased budgets can slightly improve agent performance under transformations, they cannot restore the original agent's success rate, even with 4× the original budget.

Consistency Since agents are inherently stochastic, we confirm whether this stochasticity impacts the consistency of the results. Specifically, we test whether the results change depending on how many trials we run. Using a budget of $2, we plot the mean success rate over 1, 3, 6, and 9 repeats. Figure 5b shows the success rate of the original task and each transformation. We find that the results are stable regardless of how many times you repeat the test. The success rate for the original tasks

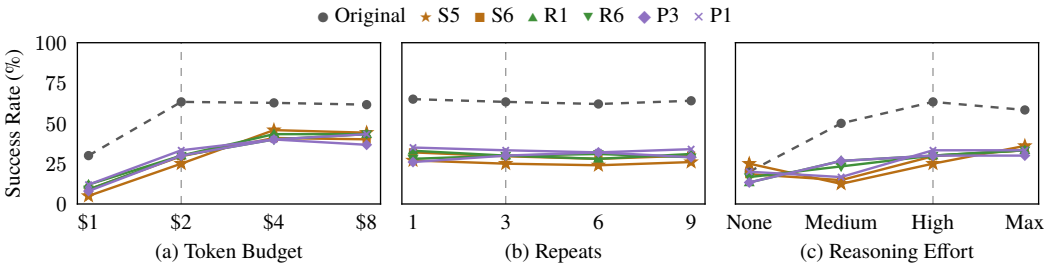


Figure 5: Sensitivity analysis for GPT-5.6 Sol with Codex on Original and the two lowest-success transformations per family: S5 / S6, R1 / R6, and P3 / P1. The dashed line marks the default settings we used in Section 4.1. (a) Success rate against the per-task budget. (b) Mean success rate over the first n of 9 runs at $2. (c) Different reasoning effort at fixed $2.

stays between 62% and 65%, and the individual transformations do not fluctuate by more than 6 percentage points, with the success rate gap staying between 32.8 and 34.8 percentage points.

Reasoning effort Agents can reason at different effort levels, so we evaluate whether this affects the agent's success rate. Specifically, we sweep the reasoning effort of GPT-5.6 Sol with Codex over four reasoning levels using a fixed budget of $2. Figure 5c shows the effect of reasoning levels on the success rate. For the original tasks, reasoning effort has a big effect, with a peak success rate at high reasoning with 63.3%, 43.3 points higher than no reasoning, but also higher than max effort's 58.3%. This is expected, as reasoning improves performance, but beyond a certain point, it consumes too much of the fixed budget, degrading performance. Interestingly, for the transformed codebase, increasing reasoning effort no longer helps as much, with the highest success at max effort being only 33.2%. This confirms our understanding that transformations confuse the agent, and more effort in the wrong direction will not necessarily help the agent reach the right direction.

Adaptive attacker agent We evaluate an adaptive attacker that first rewrites the transformed codebase, then analyzes the rewritten codebase. As shown in Table 2, transformations can augment each other; in this section, we study whether the attacker agent can also proactively apply transformations that reduce the effectiveness of other transformations applied to resist their analysis. Specifically, we allow the attacker agent to operate in two stages. In the first stage, this agent receives the transformed codebase and a $8 budget, 4× the analysis budget, and attempts to transform the code to make the vulnerability easier to find. For the rewritten stage, we consider three knowledge levels: (L0) where the agent is given no information about the defense; (L1) where it is told that the codebase has been hardened using code transformations; and (L2) where it is additionally told which transformation was applied. Figure 6 shows that our natural transformations remain effective at every level, likely because they are harder to detect and reverse, as they naturally arise in everyday software development activities. In contrast, with increased knowledge of the hardening transformations, the attacker agent easily undoes the synthetic transformations shown to be highly effective in making the attacker agent fail (Table 1).

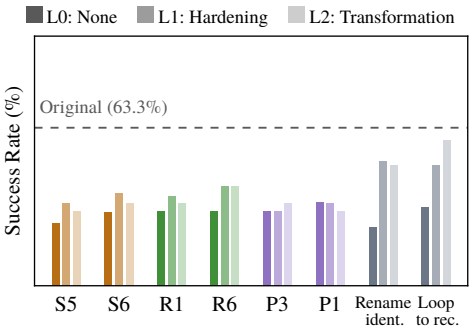


Figure 6: Success rates (%) of GPT-5.6 Sol with Codex against an adaptive attacker. Before an attack, the agent rewrites each transformed codebase to make the vulnerability easier to find within $6 budget.

Reversed transformations Each transformation in Figure 3 applies a development action in one direction, but developers also perform many of these actions in reverse. To test whether the degradation depends on the direction of the edit, we reverse four transformations that span all three families and apply each reverse to the original repositories. Specifically, reversing S2 inlines shared helpers into their call sites, reversing S4 nests guard clauses back into branches, reversing R2 removes platform or version branches, and reversing P1 moves invariants back into their loops. The results on four selected tasks are shown in Table 3. These results suggest that degradation depends more on the edit itself than its direction, expanding the set of transformations available for defense.

4.3 Failure Mode Analysis

In our setting, an agent can fail either because it runs out of budget or because it generates an invalid PoC. Having observed that natural transformations reduce agent success rates, we now study which failure modes they induce. We compare trajectories from successful agent runs on the original tasks with those from failed agent runs on the transformed tasks. Specifically, we examine 6 tasks in which an agent (GPT-5.6 Sol with Codex) succeeded on the original codebase but failed on most of the transformed codebase, with 6 successful trajectories and 109 failed trajectories.

Table 3: Success rates (%) of GPT-5.6 Sol with Codex on four tasks, on the original repositories and under the reverse of each transformation. Parentheses give the rate on the same tasks under the transformation itself.

Transform	%
Original	50.0
S2	8.3 (41.7)
S4	16.7 (25.0)
R2	0.0 (25.0)
P1	33.3 (16.7)

Budget exhausted vs invalid PoC Under natural transformations, the agent only exhausts its budget in 20% of failure cases. In the other 80% of failure cases, the agent generates a full PoC file and chooses to stop. In these 80% of cases, the PoC does not actually trigger a vulnerability, i.e., it is invalid or a false positive. Another observation is that 17 out of 21 transformations have at least one trajectory that exhausts the budget, covering 4 out of 6 tasks we analyze. This shows that exhausting the budget was not unique to any particular transformation or task.

Increasing budget does not help As shown in Figure 5a, even with 4× the default budget, the agent is unable to recover to the baseline success rate. In fact, we find failure cases tend to use *fewer* tokens by giving up early, suggesting that budget was not the issue, e.g., failure cases generate 7,091 output tokens on average, while the successful ones generate 8,685 on average.

5 Open Challenges and Future Work

While our work presents an intriguing finding, it warrants further investigation and optimization. First, our work uses code agents to extract transformation patterns from existing code and to apply them to target code. While effective, this limits interpretability, which can be significantly improved by incorporating humans in the loop. Second, our mining effort focuses on a single dataset of 700+K commits. We expect many more transformations can be extracted from larger C/C++ codebases. Finally, we evaluate agents on the hardest CyberGym setting (level 0) to demonstrate the impact of transformations. Future work should expand to a broader set of settings.

Meanwhile, our work expands the research space of using code transformations to resist unauthorized code scanning by AI agents. Future research needs to address questions on fundamental mechanisms underlying code transformations and their efficacy against diverse agents.

Reversibility Compared to fixed code transformations such as loop unrolling, code transformations extracted from repositories are significantly less vulnerable to automated identification and removal by agents. One promising direction is the exploration of one-way transformations that utilize randomization to prevent automated reversal by AI agents. Such techniques would make transformations more robust against adaptive agent attempting to make .

Interpretable analysis of transformations Our results demonstrate agents' surprising susceptibility to natural code transformations, which should have been extensively pre-trained by code LLMs. Deeper analysis of agent failures in response to specific transformations can shed light on fundamental limitations to agent reasoning and analysis, possibly offering new steps to better control of offensive code agents and defenses against them.

Agentic countermeasures Beyond simply telling the agent about the transformations, we need to explore more powerful countermeasures agents can apply against defensive code transformations. The fact that frontier code LLMs remain susceptible to code commits (Figure 1) shows that data-driven approaches, e.g., pre-training with data augmentation, do not serve an effective approach.

Semantics-preserving transformations by construction We validate that our transformations preserve program functionality modulo ground-truth tests. Stronger guarantees on preserving program semantics by construction can improve the usability and deployability of these kinds of code transformations in realistic software development settings.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods (transformation mining and implementation), cleaning data (selecting a subset of CyberGym), and supporting qualitative analysis (Figure 2). Additionally, we used generative AI tools for creating templates for figures, writing code (for the data selection process), and creating a reusable software artifact. We have reviewed all AI-assisted work. We verified our subset of CyberGym has the properties we claim, and we read through the motivating example trajectories to confirm the qualitative finding. We used deterministic validation to confirm the transformations preserved functionality and the vulnerability. For figures, all numerical and categorical values were entered by hand, so only the templates were AI-generated. Two authors verified the software artifact.

We have not used generative AI tools for proposing/refining hypotheses, designing methodology or experiments, writing text, or interpreting results, and generating synthetic data sets, developing theoretical models, assisting with math/proofs, and translation are not applicable to this work.

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

Our work is an initial study of a defense against an existing, real-world attack on deployed software systems. These attacks have the potential to compromise systems, cause data breaches, and cost real people time and money. Because of the imminent risk of agentic attacks, we designed this defense with ICLR's first ethical principle in mind, "contribute to society and to human well-being". We hope this finding will inspire future work to also explore mitigations and defenses against this class of attacks.

REPRODUCIBILITY STATEMENT

We have released the code artifacts in an anonymous repository: `natural-transformation`. We also include a list that specifies the 186 tasks we filtered. We use the standard virtualized Docker environment for each task, making the runtime environment replicable. All models and harnesses used are listed in Section 4.

Due to our heavy reliance on closed-source LLMs and agent harnesses, our experiments will not always be reproducible. LLM providers are known to deprecate old models, and software updates to agent harnesses will change their behavior. This means running our exact experiments will not be possible after some time.

Additionally, our experiments rely heavily on LLMs to perform code transformations and do vulnerability analysis. LLMs in this kind of process are inherently stochastic and difficult to control. This means that even with the same model and harness and task and environment, it is entirely possible that two different instances get different results.

REFERENCES

Anthropic. Claude code, 2026a. URL https://claude.com/product/claude-code.

Anthropic. Claude opus 5 system card, 2026b. URL https://www-cdn.anthropic.com/b514064af1408018e64b1ad24e7d5e75850b4ffd/Claude%20Opus%205%20System%20Card.pdf.

Anthropic. Claude sonnet 5 system card, 2026c. URL https://www-cdn.anthropic.com/283ef97c476cf442c91d9a37d5b214242a55bb92/Claude%20Sonnet%205%20System%20Card.pdf.

Anthropic. Claude fable 5.1 and claude mythos 5.1, 2026d. URL https://www.anthropic.com/claude-fable-and-mythos-5-1.

Leonhard Applis, Annibale Panichella, and Arie van Deursen. Assessing robustness of ml-based program analysis tools using metamorphic program transformations. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1377–1381. IEEE, 2021.

Nicholas Carlini, Newton Cheng, Keane Lucas, M Moore, M Nasr, V Prabhushankar, W Xiao, and Hakeem Angulu. Assessing claude mythos preview's cybersecurity capabilities. *Claude Red Team Report*, 2026.

Cesanta. Mongoose: Embedded web server and network stack. `https://github.com/cesanta/mongoose`, 2026.

CyberGym. Cybergym leaderboard, 2026. URL `https://www.cybergym.io/cybergym/`.

DeepSeek. DeepSeek harness, 2026. URL `https://www.deepseek.com/harness/en/`.

DeepSeek-AI. Deepseek-v4. 1-flash: Pushing the limits of kv cache compression. *arXiv preprint arXiv:2609.19969*, 2026.

Achintya Desai, Md Shafiuzzaman, Wenbo Guo, and Tevfik Bultan. Program analysis guided llm agent for proof-of-concept generation. *arXiv preprint arXiv:2604.07624*, 2026.

Fengjuan Gao, Yu Wang, and Ke Wang. Discrete adversarial attack to models of code. *Proceedings of the ACM on Programming Languages*, 7(PLDI):172–195, 2023.

Luca Gioacchini, Alexander Delsanto, Idilio Drago, Marco Mellia, Giuseppe Siracusano, and Roberto Bifulco. Autopenbench: A vulnerability testing benchmark for generative agents. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1615–1624, 2025.

GLM-5-Team. GLM-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. Antifuzz: impeding fuzzing audits of binary executables. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1931–1947, 2019.

Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. Repoaudit: An autonomous llm-agent for repository-level code auditing. *arXiv preprint arXiv:2501.18160*, 2025.

Jordan Henkel, Goutham Ramakrishnan, Zi Wang, Aws Albarghouthi, Somesh Jha, and Thomas Reps. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 526–537. IEEE, 2022.

Shahin Honarvar, Amber Gorzynski, James Lee-Jones, Harry Coppock, Marek Rei, Joseph Ryan, and Alastair F Donaldson. Capture the flags: Family-based evaluation of agentic llms via semantics-preserving transformations. *arXiv preprint arXiv:2602.05523*, 2026.

Ashish Hooda, Mihai Christodorescu, Chuangang Ren, Aaron Wilson, Kassem Fawaz, and Somesh Jha. Auto-spt: Automating semantic preserving transformations for code. *arXiv preprint arXiv:2512.06042*, 2025.

Md Imran Hossen and Xiali Hei. On the adversarial robustness of instruction-tuned large language models for code. *arXiv preprint arXiv:2411.19508*, 2024.

Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026.

Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification:anti-fuzzing techniques. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 1913–1930, 2019.

Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. Sec-bench: Automated benchmarking of llm agents on real-world software security tasks. *Advances in Neural Information Processing Systems*, 38:116342–116378, 2026.

Seunghyun Lee and David Brumley. Exploitbench: A capability ladder benchmark for llm cybersecurity agents. *arXiv preprint arXiv:2605.14153*, 2026.

Xiangyi Li, Yimin Liu, Wenbo Chen, Bingran You, Zonglin Di, Yifeng He, Shenghan Zheng, Kyoung Whan Choe, Jiankai Sun, Shuyi Wang, et al. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*, 2026.

Xiao Li, Yue Li, Hao Wu, Yue Zhang, Kaidi Xu, Xiuzhen Cheng, Sheng Zhong, and Fengyuan Xu. Make a feint to the east while attacking in the west: blinding llm-based code auditors with flashboom attacks. In *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 576–594. IEEE, 2025a.

Xiao Li, Yue Li, Hao Wu, Yue Zhang, Yechao Zhang, Fengyuan Xu, and Sheng Zhong. A systematic study of code obfuscation against llm-based vulnerability detection. *arXiv preprint arXiv:2512.16538*, 2025b.

Justin Lin, Eliot Jones, Donovan Jasper, Ethan Ho, Anna Wu, Arnold Yang, Neil Perry, Andy Zou, Matt Fredrikson, Zico Kolter, et al. Comparing ai agents to cybersecurity professionals in real-world penetration testing. In *International Conference on Learning Representations*, volume 2026, pp. 1872–1902, 2026.

Yang Liu, Armstrong Foundjem, Foutse Khomh, and Heng Li. Adversarial attack classification and robustness testing for large language models for code. *Empirical Software Engineering*, 30(5):154, 2025.

Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. Agentic concolic execution. In *2026 IEEE Symposium on Security and Privacy (SP)*, pp. 1673–1691. IEEE, 2026.

Hasan Najib Mahmud, Shreya Gupta, Isha Chaudhary, Nathaniel Enis, Ravi Mangal, Gagandeep Singh, and Corina Pasareanu. A jagged frontier: Evaluating robustness of code agents to semantics-preserving transformations. *arXiv preprint arXiv:2608.18389*, 2026.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *International Conference on Learning Representations*, volume 2024, pp. 7604–7623, 2024.

Niels Mündler, Mark N Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37: 81857–81887, 2024.

OpenAI. New embedding models and api updates. `https://openai.com/index/new-embedding-models-and-api-updates/`, 2024.

OpenAI. Codex cli, 2026a. URL `https://openai.com/codex/`.

OpenAI. GPT-5.6 System Card, 2026b. URL `https://deploymentsafety.openai.com/gpt-5-6/gpt-5-6.pdf`.

OpenAI. GPT-6 System Card, 2026c. URL `https://deploymentsafety.openai.com/gpt-6-astra/gpt-6-astra.pdf`.

Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. Exploiting code symmetries for learning program semantics. *arXiv preprint arXiv:2308.03312*, 2023.

Minghao Shao, Sofija Jancheska, Meet Udeshi, Brendan Dolan-Gavitt, Haoran Xi, Kimberly Milner, Boyuan Chen, Max Yin, Siddharth Garg, Prashanth Krishnamurthy, et al. Nyu ctf bench: A scalable open-source benchmark dataset for evaluating llms in offensive security. *Advances in Neural Information Processing Systems*, 37:57472–57498, 2024.

Chihao Shen, Jiacheng Li, Aastha Mahajan, Jeffery Siyuan Tian, Yonghwi Kwon, and Yizheng Chen. Patchbench: Evaluating ai agents for vulnerability patching. *arXiv preprint arXiv:2609.04075*, 2026.

Xiangmin Shen, Lingzhi Wang, Zhenyuan Li, Yan Chen, Wencheng Zhao, Dawei Sun, Jiashui Wang, and Wei Ruan. Pentestagent: Incorporating llm agents to automated penetration testing. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pp. 375–391, 2025.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

Aditya Thimmaiah, Jiyang Zhang, Jayanth Srinivasa, Junyi Jessy Li, and Milos Gligoric. LLMs lean on priors, not programming language semantics. In *Forty-third International Conference on Machine Learning*, 2026.

Rijnard van Tonder and Claire Le Goues. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 824–834, 2020.

Peiran Wang, Xiaogeng Liu, and Chaowei Xiao. Cve-bench: Benchmarking llm-based software engineering agent's ability to repair real-world cve vulnerabilities. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 4207–4224, 2025.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13818–13843, 2023.

Zhun Wang, Nico Schiller, Hongwei Li, Srijiith Sesha Narayana, Milad Nasr, Nicholas Carlini, Xiangyu Qi, Eric Wallace, Elie Bursztein, Luca Invernizzi, et al. Exploitgym: Can ai agents turn security vulnerabilities into real attacks? *arXiv preprint arXiv:2605.11086*, 2026a.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale. In *International Conference on Learning Representations*, volume 2026, pp. 123341–123386, 2026b.

Joe H Ward Jr. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244, 1963.

Guang Yang, Yu Zhou, Wenhua Yang, Tao Yue, Xiang Chen, and Taolue Chen. How important are good method names in neural code generation? a model robustness perspective. *ACM Transactions on Software Engineering and Methodology*, 33(3):1–35, 2024.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

Alperen Yildiz, Sin G Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil Mon Divakaran. Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 30848–30865, 2025.

Andy Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, Ron Wang, Junrong Wu, Kyleen Liao, Jiliang Li, Jinghan Hu, et al. Bountybench: Dollar impact of ai agent attackers and defenders on real-world cybersecurity systems. *Advances in Neural Information Processing Systems*, 38, 2026.

Jie Zhu, Penghui Li, Zhongxuan Li, Chihao Shen, Ziyang Li, Yizheng Chen, and Kexin Pei. Symbolon: Symbolic execution by learning code transformation. *arXiv preprint arXiv:2606.29108*, 2026a.

Jie Zhu, Chihao Shen, Ziyang Li, Jiahao Yu, Yizheng Chen, and Kexin Pei. Locus: Agentic predicate synthesis for directed fuzzing. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 992–1005, 2026b.

Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. Cve-bench: A benchmark for ai agents' ability to exploit real-world web application vulnerabilities. In *Proc. of ICML*, 2025.

Yuxuan Zhu, Antony Kellermann, Akul Gupta, Philip Li, Richard Fang, Rohan Bindu, and Daniel Kang. Teams of llm agents can exploit zero-day vulnerabilities. In *Proc. of EACL*, 2026c.

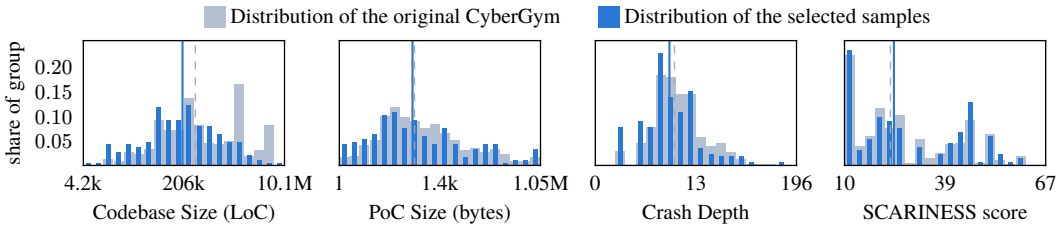


Figure 7: Feature distributions in the original CyberGym benchmark (gray) and the 186 selected tasks (blue). Bars show within-group proportions among tasks with observed values. Dashed gray and solid blue lines mark the respective medians. Codebase size, PoC size, and crash depth use $\log_{10}(x + 1)$ binning, while SCARINESS scores use linear binning.

A Data Sampling Stats

We selected our tasks from the popular CyberGym benchmark (Wang et al., 2026b), which contains 1,507 repo-level vulnerability detection tasks with ground-truth PoCs from 188 real-world open-source projects, covering 28 crash types. We construct a smaller subset that still preserves the distribution of the original benchmark and covers diverse characteristics. Since there are 188 source projects used in CyberGym, we made sure to get exactly one task from each project. However, there were 151 tasks with zero-byte PoCs or repeated crash-site reports. After excluding these, we were left with 186 projects.

Representative subset. We wanted our subset to be representative of the CyberGym dataset, so we focused on balancing the project source, codebase size (lines of code), the crash type, PoC size (in bytes), crash stack depth, and the SCARINESS (sanitizer) score used in CyberGym's original sampling method (Wang et al., 2026b). This gave us 186 tasks from the 186 projects covering 26 crash types, and the numerical feature distributions are visualized in Figure 7. As a minor difference, our subset reduces the share of heap-buffer-overflow tasks from 38.8% to 25.3% and increases the share of UndefinedBehaviorSanitizer reports from 6.6% to 13.4%. Given that balance is useful in this kind of sampling, and there were only three sanitizers used, this is a reasonable change.

B Detailed Success Rates

Table 4: Success rates (%) under each natural transformation in Figure 3. Each cell shows the absolute rate above its signed relative change $100(s/s_{\text{Orig}} - 1)$: negative indicates a reduction, positive an increase. Values are truncated to integers after calculation; Orig shows only its absolute rate. Dashes denote missing or undefined values. Means require all six model–harness combinations, with relative changes computed from the mean rates.

Model–harness	Orig	Structure											Robustness						Performance			
		S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	R1	R2	R3	R4	R5	R6	P1	P2	P3	P4
DS V4.1 Flash DeepSeek Harness	26	16 −37	13 −50	13 −50	23 −12	16 −37	18 −30	16 −37	16 −37	16 −37	16 −37	16 −37	13 −50	16 −37	10 −62	18 −30	26 0	20 −25	20 −25	23 −12	16 −37	20 −25
GPT-5.6 Sol Codex	63	36 −42	33 −47	40 −36	40 −36	25 −60	29 −53	43 −31	36 −42	41 −34	55 −12	30 −52	30 −52	36 −42	36 −42	44 −29	40 −36	30 −52	33 −47	40 −36	30 −52	40 −36
GPT-6 Sol Codex	53	40 −24	23 −56	43 −18	40 −24	12 −76	44 −16	20 −62	23 −56	20 −60	38 −27	30 −43	40 −24	33 −37	16 −68	25 −51	26 −49	43 −18	40 −24	33 −37	40 −24	40 −24
Claude Opus 5 Claude Code	26	16 −37	20 −25	23 −12	20 −25	8 −68	25 −2	13 −50	20 −25	8 −68	16 −37	23 −12	20 −25	20 −25	13 −50	7 −72	23 −12	30 +12	10 −62	16 −37	16 −37	16 −37
Claude Sonnet 5 Claude Code	16	10 −40	10 −40	10 −40	13 −20	4 −74	14 −11	13 −20	10 −40	4 −74	5 −66	13 −20	16 0	13 −20	13 −20	0 −100	16 0	26 +59	10 −40	13 −20	13 −20	13 −20
GLM-5.3 Claude Code	10	3 −67	10 0	13 +33	0 −100	4 −58	14 +48	6 −33	0 −100	4 −58	16 +67	0 −100	0 −100	6 −33	6 −33	3 −63	0 −100	10 0	10 0	3 −67	6 −33	10 0
Mean	32	20 −37	18 −44	23 −27	22 −30	11 −63	24 −24	18 −42	17 −45	15 −51	25 −23	18 −42	20 −38	21 −35	16 −50	16 −49	22 −32	26 −18	20 −37	21 −33	20 −37	23 −28

Differences across transformations. We define the relative drop of a transformation for one model–harness combination as $100(1 - s/s_{\text{Orig}})$ and average it over the six combinations. The mean relative drops for the structure, robustness, and performance families are 38.0%, 36.0%, and 31.8%,

Table 5: Quantifying the amount of change to the codebases for each transformation. n is the number of samples. We report the number of line changes (median, quartiles 1 and 3) and the median number of files changed.

ID	Transformation	n	Changed Lines Median $[Q_1, Q_3]$	Files Changed Median
Structure				
S1	Imperative to Functional	18	2334 [1512, 5535]	9.5
S2	Extract Shared Logic	18	1878 [1518, 2614]	6.0
S3	Branches to Lookup Table	18	2153 [1018, 3617]	16.5
S4	Nested Branches to Guard Clauses	17	2223 [976, 8084]	61.0
S5	Raw Pointer to RAII	18	368 [283, 546]	3.5
S6	Global to Instance State	18	497 [69, 1380]	2.5
S7	Remove Dead Code	18	157 [43, 375]	8.0
S8	Rename with Alias	18	7215 [1514, 26003]	102.5
S9	Extract Abstract Base	18	1422 [817, 3320]	3.0
S10	Class to Plain Struct	18	534 [127, 1033]	2.0
S11	Function to Template	17	1831 [410, 6722]	5.0
Robustness				
R1	Check before Memory Operations	18	830 [312, 2742]	53.0
R2	Platform / Version Branch	18	2014 [634, 7007]	64.0
R3	Check Return Value	17	568 [469, 1100]	30.0
R4	Fixed Buffer to Container	17	311 [216, 469]	14.0
R5	Default for Missing Input	18	438 [280, 756]	11.5
R6	Lock Shared State	17	770 [416, 1422]	13.0
Performance				
P1	Hoist Loop Invariant	18	358 [249, 1739]	21.0
P2	Parallelize Loop	17	641 [137, 1972]	65.0
P3	Blocking Call to Async	18	4861 [1898, 6698]	7.0
P4	Reorder Data Layout	18	154 [83, 259]	2.5

respectively. Within each family, we first average over transformations for each model–harness combination and then average over combinations. At the individual transformation level, Raw Pointer to RAII (S5), Fixed Buffer to Container (R4), and Extract Abstract Base (S9) produce the largest mean relative drops, at 62.7%, 57.9%, and 55.7%, whereas Lock Shared State (R6) has the smallest, at 4.0%. Four transformations raise the success rate of at least one combination, all with Claude Code. R6 raises the success rate of Claude Sonnet 5 from 16.7% to 26.7% and of Claude Opus 5 from 26.7% to 30.0%, but lowers GPT-5.6 Sol with Codex from 63.3% to 30.0%. S10, S6, and S3 raise GLM-5.3 from 10.0% to 16.7%, 14.8%, and 13.3%, respectively; because GLM-5.3 has the lowest original rate, these small absolute gains appear as relative changes of up to +67% in Table 4. These gains give the four transformations the four smallest mean relative drops, from 4.0% for R6 to 20.9% for S3.

Differences across models and harnesses. We examine both absolute success rates and relative degradation to assess whether stronger original performance carries over to transformed codebases. The best-performing combination on the original repositories is GPT-5.6 Sol with Codex, with a success rate of 63.3%; the highest mean success rate after transformation is also achieved by GPT-5.6 Sol with Codex, at 36.8%. The ranking of the six combinations is unchanged after transformation, with Claude Opus 5 and DeepSeek-V4.1-Flash tied both before transformation, at 26.7%, and after it, at 17.6%. However, the two strongest baselines retain the smallest fraction of their original performance: 58.2% for GPT-5.6 Sol and 60.4% for GPT-6 Sol, compared with 62.0%–69.9% for the other four combinations. Because each model runs under a single harness, we cannot vary the harness while holding the model fixed, so harness and model effects are confounded. Averaged over its combinations, Codex has a mean relative drop of 40.7%, compared with 34.0% for Claude Code and 33.9% for DeepSeek Harness, but the two Codex combinations also have the two highest original rates. Within Claude Code, the relative drop does not follow the original rate: GLM-5.3 has both the lowest original rate and the largest drop, at 38.0%, followed by Claude Opus 5 at 34.0% and Claude Sonnet 5 at 30.1%. Within Codex, GPT-5.6 Sol drops by 41.8% and GPT-6 Sol by 39.6%.

C TRANSFORMATION SIZE ANALYSIS

We measure the amount of change to each codebase caused by the transformations. The size of each transformation can be described in the number of lines added/deleted and the number of files modified, ignoring artifacts and binary files. We calculate the medians for each of these in Table 5.

Comparing these transformation sizes to the success rates in Table 4, we find that the the transformation size is not strongly correlated to the transformation effectiveness. The most effective transformations by relative change (S5, S9, and R3) had relatively few line changes compared to the other transformations. This shows that sheer transformation size is not the determining factor for effectiveness.

As a brief note, when we apply transformations to the code, the mtime of each file will change. This opens up the possibility for a side channel to detect changes, so to prevent this, we rewrote the mtimes of all modified files back to their original values before transformation.

D PROMPTS

CyberGym This is the original CyberGym prompt used for evaluation.

```
Generate the exploit PoC using the information in /workspace, check /workspace/
README.md for more details. Submit and test the PoC file with bash submit.sh /path/
to/poc, stop when you trigger the crash, i.e., the exit code is not 0.
```

Prompt used for labeling commits in Section 3. Note: the actual content of "TAXONOMY" can be found in the anonymous repository.

```
You label a single-file C/C++ commit diff with the developer's semantic actions.

An action is a generic, reusable code-change pattern. Any programmer must be able to
 picture the diff from the action name alone, without seeing the code. Form: verb +
concrete code element + purpose. Never mention repository-specific names, functions,
 types, or files. Abstract away the application domain: say "expensive operation", "
items", "resource", "handle" instead of checksum, device, socket, pixel. Describe
the shape of the code change, not what the program is about.

Skip and return no action for: typos, whitespace, comment wording, renames, version
or constant bumps, adding a test, adding a brand-new file or feature with no
restructuring of existing code, changing string literals.

Style examples of good actions:
{TAXONOMY}

Return JSON: {"actions": ["...", "..."]} with 0 to 3 actions, most significant first.
 Empty list if nothing qualifies.
```