Post-hoc Attention Steering of Large Language Models for Robust Code Understanding under Obfuscation

Xiaokai Rong
University of Texas at Dallas
Dallas, USA
xiaokai.rong@utdallas.edu

Aashish Yadavally
University of Central Florida
Orlando, USA
aashish.yadavally@ucf.edu

Tien N. Nguyen✉
University of Texas at Dallas
Dallas, USA
tien.n.nguyen@utdallas.edu

Abstract

Code obfuscation is widely used in software systems and malware to conceal program logic and hinder analysis, posing significant challenges for both human developers and automated tools. While large language models (LLMs) have shown strong capabilities in code understanding, our preliminary study shows that LLM performance significantly degrades on obfuscated code, suggesting a reliance on superficial lexical cues rather than deep semantic reasoning. To address this limitation, we propose CodeSteer, a novel *attention steering* approach that reallocates model attention toward semantically relevant program elements, including backward slices for output prediction and control-flow paths for execution reasoning. Our method integrates lightweight program analysis with inference-time attention steering to guide LLMs toward the core input-to-output dependencies of a program. Experiments across multiple models and datasets demonstrate that CodeSteer significantly improves performance on obfuscated code, often recovering comparable accuracy to the level of unobfuscated programs. We also show CodeSteer's practical utility through a case study on buffer overflow detection, highlighting its potential for malware/vulnerability analysis and reverse engineering of obfuscated code.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Software and its engineering**;

Keywords

Keywords: AI4SE, model attention steering, code reasoning

ACM Reference Format:
Xiaokai Rong, Aashish Yadavally, and Tien N. Nguyen. 2026. Post-hoc Attention Steering of Large Language Models for Robust Code Understanding under Obfuscation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3837556

1 Introduction

Malware frequently leverages code obfuscation to disguise its attack mechanisms and internal logic, posing challenges for program analysis and reverse engineering. Code obfuscation is also widely used in software to protect intellectual property and conceal business logic. Obfuscation techniques transform a program into a semantically equivalent form that is intentionally difficult for humans to read or understand. Typical transformations include identifier renaming, control-flow flattening, dead-code insertion, call indirection, etc. While these transformations preserve code functionality, they significantly reduce readability and increase the cognitive effort required for understanding program behavior [5, 6, 20, 23].

Prior research has shown the substantial impact of obfuscation on human comprehension. Nguyen *et al.* [17] reported that obfuscated code significantly reduces humans' accuracy in predicting program outputs. Their study further revealed an interesting cognitive phenomenon: when variable names are adversarially renamed to misleading identifiers, developers shift from fast, memory-based reasoning (System 1) to slower analytical reasoning (System 2) [17]. Meaningful identifier names, thus, serve as cognitive shortcuts for code comprehension, and removing those cues forces humans to rely on stepwise reasoning over program flows.

Large language models (LLMs) have demonstrated impressive capabilities in tasks such as code generation, summarization, vulnerability detection, and program reasoning [2, 9, 11, 14, 18]. Most evaluations of these models are conducted on naturally written source code where identifier names, comments, and common coding patterns provide strong semantic signals. With the rapid emergence of LLMs for code understanding and generation, an important open question arises: *whether LLMs can maintain their reasoning capabilities when these signals are intentionally disrupted through obfuscation?* Answering this question has important implications for software security and analysis. If LLMs remain robust to obfuscation, they could become powerful tools for *analyzing and reverse engineering malware*. Conversely, if LLMs are confused by obfuscated code in a manner similar to humans, the effectiveness of current obfuscation techniques may extend to AI-based analysis.

Our preliminary investigation reveals that LLMs are indeed significantly affected by obfuscation. Across several models, the accuracy of program output prediction decreases substantially when programs are obfuscated, even though the semantics of the code remain unchanged. This is consistent with the findings by Nikiema *et al.* [19] as they reported a statistically significant performance decline as obfuscation complexity increases. This observation suggests that modern LLMs rely heavily on superficial lexical cues such as identifier names and familiar coding patterns rather than purely reasoning on program semantics. This is particularly concerning in security-critical settings, e.g., *malware analysis*. The inability of LLMs to robustly reason about obfuscated code may therefore limit their applicability in software security. Addressing this gap is crucial for *enabling reliable AI-assisted analysis of adversarial code*.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837556

Improving LLMs' code understanding on obfuscated code requires mechanisms that guide the models to focus on the program elements that are truly relevant for the task at hand, e.g., program output and execution prediction, rather than the above superficial lexical cues. A key challenge is that modern LLMs process code primarily as token sequences, and their attention mechanisms may distribute focus broadly across tokens that contain superficial lexical cues rather than the important statements that mainly contribute to decide the program's behavior. When those cues are removed or misleading as in obfuscated code, the model may fail to concentrate on crucial semantic dependencies.

This motivates us to propose CodeSteer, which explores **attention reallocation** (or steering) as a training-free vehicle to guide LLM reasoning. Attention steering has emerged as a practical mechanism for influencing the reasoning trajectory of transformer models by biasing attention toward important tokens [25, 26]. CodeSteer uses lightweight static program analysis to construct a soft relevance prior over code tokens including static slicing, data/control dependencies, and calling relations. This prior helps a model to pay attention to important tokens in the absence of lexical cues. To achieve that, CodeSteer aims to steer LLM attention toward these semantically critical program elements so that the model's reasoning process aligns more closely with the true data and control flows in the program. By reallocating attention toward these semantically meaningful structures, the model can better reconstruct the logical relationships embedded in the program despite the absence of meaningful lexical cues.

We evaluate CodeSteer with multiple LLMs and code understanding datasets. Our results show that attention reallocation significantly improves LLM output prediction and execution trace prediction performance on obfuscated code, recovering much of the accuracy lost due to obfuscation, even with the accuracy comparable to that obtained on the original unobfuscated code.

Interestingly, in a few scenarios, the steered models even outperform their accuracy on the original programs. A deeper analysis reveals that the *steering process induces longer and more meaningful reasoning traces in the models*, suggesting that they allocate more tokens to intermediate reasoning steps. This behavior resembles the *cognitive shift observed in humans*: once lexical shortcuts such as meaningful names are removed, both humans and LLMs appear to rely more heavily on deliberate reasoning processes analogous to System 2 thinking as in humans [17]. Consequently, the model may analyze program semantics more thoroughly than when relying on superficial cues in the original code. Importantly, we showcase the practical utility of CodeSteer through a case study on ***buffer overflow vulnerability analysis***, underscoring its promise for malware analysis. This paper makes the following contributions:

(1) Our findings have important **implications** for both software security and model code intelligence. First, they suggest that LLMs are vulnerable to obfuscation in a way similar to humans, implying that *existing obfuscation techniques may remain effective against LLMs*. Second, our results show that targeted attention steering can mitigate this limitation and substantially improve reasoning over obfuscated code. Thus, *steered models are useful for understanding obfuscated code in malware analysis*.
(2) We propose a novel attention reallocation framework that relies on light-weight static priors including static slicing, data/control dependencies, and calling relations. It guides a model to focus on semantically relevant program elements for better comprehension of obfuscated code in dynamic behaviors.
(3) We demonstrate through extensive experiments across multiple LLMs and datasets that steering significantly mitigates the negative impact of obfuscation and improves model performance.

Example 1: Non-obfuscated program

```java
public class BinomialCoefficient {
    public static void main(String[] args) {
        int n = 3;
        int k = 2;
        int coefficient = binomialCoefficient(n, k);
        System.out.println(coefficient);
    }
    public static int binomialCoefficient(int n, int k) {
        int res = 1;
        if (k > n - k)
            k = n - k;
        for (int i = 0; i < k; ++i) {
            res *= (n - i);
            res /= (i + 1);
        }
        return res;
    }
}
```

Representative model generation

"The `binomialCoefficient` function calculates the binomial coefficient $C(n,k)$, which is the number of ways to choose k elements from a set of n elements. The function uses an iterative approach to calculate the coefficient. It first checks whether k is greater than $n-k$ and, if so, sets $k = n-k$ to reduce the number of multiplications. The result is initialized to 1, and for each i the function multiplies by $(n-i)$ and divides by $(i+1)$. The expected output of the program is 3 because the binomial coefficient $C(3,2)$ is 3. Therefore, the exact console output is 3."

Figure 1: Model reasoning on non-obfuscated code

2 Motivating Example

Non-obfuscated Code. We start from a Java program that computes the binomial coefficient $C(3,2)$ (Figure 1). We first modified that program by using an obfuscation tool to rename the variables, add dead code, and change the control flow (Figure 2). Both programs produce the same output, yet our goal is to investigate whether obfuscation changes *how* the model gets to that answer.

We used the following to prompt Qwen2.5 7B on the original code: *"You will analyze the Java program provided below. First, explain how you will predict the output for the code snippet. Then, simulate it as a compiler would and print the exact console output."*

Color Codes: The colored emphasis is added by us to highlight the reasoning pattern expressed in the model generation: semantic retrieval / use of pretrained algorithm knowledge; control-flow inspection and simplification of obfuscation artifacts; state tracking and arithmetic simulation; final output prediction.

The output from GPT-4 was shown in Figure 1. As seen, the model exploits the descriptive method name `binomialCoefficient` and answers in a retrieval-style manner, immediately mapping the task to a familiar mathematical formula and *computing the output via the formula*. That is, the original version enables algorithm recognition, encourages a *semantic shortcut*. The model might fall into the scenario of not spending much space reasoning branches or narrating loop iterations; instead, it quickly maps the method

Example 2: Fully obfuscated program

```java
public class C_7a2cbda3 {
    public static void main(String[] args) {
        if (((3 * 3) - 9) != 0) {
            int _obf_t5_main_405592 = 0;
            _obf_t5_main_405592++;}
        if (((2 * 2) + 1) > 0) {
            int _obf_t5_main_405592_m = 0;
            _obf_t5_main_405592_m += 0;
        } else {
            int _obf_t5_main_405592_e = 1;
            _obf_t5_main_405592_e -= 1;}
        int _obf_t5_main_405592_h = (1 ^ 1);
        if (_obf_t5_main_405592_h != 0) {
            _obf_t5_main_405592_h += 1;}
        int n = 3;
        int k = 2;
        int coefficient = m_O0lI1(n, k);
        System.out.println(coefficient);}
    public static int m_O0lI1(int n, int k) {
        int _obf_t5_m_O0lI1_591095 = 0; _obf_t5_m_O0lI1_591095 += 0;
        int res = 1;
        if (k > n - k)
            k = n - k;
        for (int i = 0; i < k; ++i) {
            res *= (n - i);
            res /= (i + 1);}
        return res;}
}
```

Representative model generation

"The main method starts by evaluating the condition `((3 * 3) - 9) != 0`. This simplifies to $9 - 9 \neq 0$, i.e., $0 \neq 0$, so the first block is skipped. The next condition `((2 * 2) + 1) > 0` becomes $5 > 0$, so the second block executes, although it has no effect on the final value. The variable `_obf_t5_main_405592_h` is initialized as `1 ^ 1`, which is 0, so the following guarded block is also skipped.
The variables n and k are initialized to 3 and 2, and `m_O0lI1(n, k)` is called. Inside `m_O0lI1`, the result `res` starts at 1. Since $k > n - k$ ($2 > 1$), the method updates k to $n - k = 1$. The loop now runs once. For $i = 0$, `res *= (n-i)` changes `res` from 1 to 3, and `res /= (i+1)` keeps it at 3. The method returns 3, which is assigned to `coefficient`. The program prints 3."

Figure 2: Model reasoning on fully obfuscated code

name `binomialCoefficient` to a known concept and explains the output at the formula level. Thus, the explanation is primarily driven by recognizing the lexical cue in `binomialCoefficient` as a familiar combinatorial routine, with relatively little attention devoted to reasoning about statements and branch/loop conditions.

Obfuscated Code. Next, we used the same prompt on the obfuscated version with renamed variables, dead code, and changed control flow (Figure 2). As seen in its textual output, the descriptive cue disappears and the surrounding code becomes cluttered with opaque predicates and no-op statements. Once the descriptive method name is removed and obfuscation artifacts are inserted, the model then appears to shift into an execution-style mode and the generation changes noticeably. Instead of jumping directly to a formula-level explanation as earlier, the model *walks through opaque predicates, dead blocks, and arithmetic updates in each iteration.* The answer is correct, but the *path* to the answer is now explicitly trace-based, even though these steps are unnecessary once the algorithm is recognized. This behavior parallels the cognitive shift observed in humans: when lexical shortcuts (meaningful identifier names) are removed, LLMs rely more heavily on deliberate analytical reasoning. This phenomenon is similar to humans' behaviors in which human developers switch to System 2's deliberate thinking when the lexical patterns in the code disappear [17].

3 Preliminary Study

We first conducted a preliminary study that establishes our empirical premise on *whether semantically preserving obfuscation reduces LLM performance on program output prediction even when the underlying program behavior is unchanged.* We intentionally restrict this preliminary study to the *output-prediction* setting. If a model produces an incorrect output for an obfuscated program, it indicates that the model has failed to capture the correct execution.

Data Collection and Benchmark Selection. Our preliminary study uses Java benchmark instances derived from two benchmarks: HumanEval [27] and CruxEval [10]. We chose them because they provide *controlled, executable benchmarks* for evaluation with *reliably built ground truth for dynamic reasoning tasks* including output prediction and execution-trace prediction. For HumanEval-X [27], we use the Java split containing 164 tasks. For CruxEval-X [10], we start from the benchmark's output-prediction setting and retain only those Java-translated instances whose benchmark harnesses execute reliably and from which output-prediction supervision can be derived directly. After filtering, the evaluation corpus contains 164 HumanEval-X snippets and 698 CruxEval-X snippets.

Note that the effective size of the dataset is determined not only by the number of snippets, but also by how many output-checking cases can be derived from each snippet, i.e., how many usable seed checks each snippet provides and how many variants of checks can be generated from those seeds. HumanEval-X contains fewer snippets overall, but many of its benchmark harnesses include multiple boolean checks. Each such check can serve as a seed assertion and can further produce additional validated counterfactual mutations, resulting in relatively large case packs per snippet.

Counterfactual Output-Prediction Case Construction. We derive output-prediction labels automatically from the original benchmark harnesses rather than annotating them manually. For each retained snippet, we construct a validated counterfactual case pack starting from seed true-cases extracted from the harness. In HumanEval-X, these seeds correspond to boolean expressions inside `List<Boolean> correct = Arrays.asList(...)`; in CruxEval-X, they correspond to `assert(...)` predicates. Each seed is *re-executed* on the actual program, and seeds that do not evaluate to `true` are discarded as unstable. Starting from the surviving seeds, we generate candidate false-cases by applying small mutations to the original checks. Following mutation testing, our mutation operation types include `expected_value_mutation`, `threshold_mutation`, `literal_perturbation`, `argument_perturbation`, `comparator_flip`, and `negation_fallback`. Each mutated case is retained only if it executes successfully and evaluates to `false`. Finally, we remove trivial literal cases, including `assert true/false`, `assert stringLiteral`, as the model can easily guess the output after mutations. We keep only non-trivial cases that are validated through program execution. This procedure yields 1,922 validated output-prediction cases for HumanEval-X and 1,378 for CruxEval-X. Averaged over snippets, this corresponds to 11.72 cases per HumanEval snippet and 1.97 cases per CruxEval-X snippet. Although CruxEval-X contributes more programs, HumanEval-X contributes substantially larger case packs per program.

Obfuscation Augmentation and Semantic Validation. We augment each clean Java snippet with semantics-preserving source-to-source obfuscated variants. Our transformation space is organized into

Table 1: Preliminary output-prediction results. Obfuscated results aggregate across all 4 obfuscation types.

		P@1 (Orig.) (%)	P@1 (Obf.) (%)
Qwen2.5-7B	HumanEval-X	76.49	64.01
	CruxEval-X	83.11	71.40
DeepSeek-6.7B	HumanEval-X	77.74	68.29
	CruxEval-X	78.98	66.93
Qwen2.5-14B	HumanEval-X	94.63	85.31
	CruxEval-X	92.14	86.48
DeepSeek-V2-Lite	HumanEval-X	90.31	80.18
	CruxEval-X	91.56	81.11

four representative families: *identifier renaming*, *dead-code injection*, *control-flow flattening*[16], and *call indirection*. To ensure that obfuscation does not change program semantics, each transformed variant is validated against the original benchmark harness and retained only if the observable behavior matches the original, including task-relevant labels under the harness.

Experimental Procedure and Metric. For each individual case in a case pack, we prompt the model separately with the corresponding program and ask it to return the answer containing a binary prediction, `T` or `F`. Each case is evaluated with *three independent runs* under the same prompting and decoding regime. A case is counted as *passed* at k if at least one of the first k independent responses for that case matches the ground-truth label. The reported score is the proportion of cases that pass within k attempts.

Preliminary Results. In Table 1, the obfuscated condition is consistently lower than the original condition at Pass@1. This pattern appears on both HumanEval-X and CruxEval-X, indicating that semantics-preserving obfuscation degrades output-prediction performance. This result has the following implications:

(1) LLM performance **degrades substantially** when code obfuscation is present. In particular, their ability to reason about program behavior becomes significantly weaker on obfuscated programs, mirroring the confusion experienced by humans when reading obfuscated code as reported by Nguyen *et al.* [17].

(2) The decline in accuracy arises because obfuscation disrupts surface-level lexical and structural cues that LLMs often exploit when analyzing non-obfuscated code, (see motivating example).

(3) These findings motivate our next question in this work: whether inference-time attention steering can redirect an LLM toward semantically relevant program elements and thereby (partially) recover the performance lost under obfuscation. This would help improve its code understanding, leading to better malware analysis.

4 Post-Hoc Attention Steering

In this work, we evaluate models' capability in two fundamental tasks in code understanding: *execution prediction* and *output prediction*. Execution prediction requires reasoning about which statements are executed for an input, while output prediction requires the computation of variables' values, deciding the final result.

To guide the model without training/fine-tuning, CodeSteer constructs a static relevance prior over code tokens using lightweight program analysis. These signals are not intended to presolve these above tasks that require dynamic reasoning; rather, the key intuition is that these static signals provide a soft bias that helps the model allocate more attention to code regions that are structurally connected to the program behavior being reasoned about when the lexical cues disappear in the source code.

Concretely, CodeSteer uses control-flow and dependence information to estimate semantic relevance. Control-flow information captures how statements may be ordered and guarded during execution, while dependence information captures how values may flow through assignments, variables, calls, predicates, and return expressions. Based on this analysis, CodeSteer maps relevant program elements to their corresponding source spans and then to prompt-token positions. The resulting token set forms a soft attention prior used by the steering algorithm during decoding, while the full program remains available in the context. Importantly, this prior does not mask irrelevant code or remove obfuscation artifacts. In fact, this prior is static, it may be incomplete or noisy for concrete runtime tasks; the LLM must still reason about branch outcomes, loop iterations, intermediate values, and final outputs during generation.

4.1 Program Slicing to Build Steering Priors

While the statements along the static control flow from the input are straightforward to identify, let us explain the procedure that we use static backward slicing to identify the crucial tokens for output prediction. First, the **steering prior** is a prompt-aligned relevance distribution derived from our program dependence approximation. It provides a stable token-level signal indicating which parts of the program are likely to affect the output prediction.

Data Pre-processing. For a program, we first identify a set of observable *task sinks* S, defined as the expressions whose evaluated values determine the task outcome. As a correctly predicted output is crucial for dynamic reasoning, S consists of the expressions passed to `System.out.println(...)` within the `main` method. In harness-based benchmarks, S consists of the boolean predicates that encode correctness, such as assertions or equality checks (e.g., `s.vowelsCount("ACEDY") == 3`). For each sink $s \in S$, we identify the program elements that directly contribute to its value. If the value of s depends on a call to a user-defined method, we include the corresponding call result and the associated return expressions as *output seeds*, and designate that callee as a *target method*. Otherwise, the enclosing method of s is treated as the target method. When multiple sinks or multiple target methods are present, we retain all of them and aggregate their reachable dependencies, rather than imposing a single sink. This ensures that the subsequent analysis captures all elements that may affect the observable outcome.

Building Static Program Slices. Starting with a target method and any helper methods that can be reached from them, we build a static dependence graph using Joern. We only keep the parts of the graph that are necessary for steering, and represent it as

$$G = (V, E_{\text{data}} \cup E_{\text{ctrl}} \cup E_{\text{call}}), \tag{1}$$

where the nodes correspond to program elements such as assignments, variable uses, control conditions, return expressions, function parameters, method calls, and output-related predicates. The edges capture different types of relationships: data edges represent how values flow between variables (i.e., definition–use relations), control edges represent how conditions (e.g., if-statements) determine if other statements execute, and call edges connect arguments to parameters and link return values back to where they are used.

Identifying crucial tokens for a steering prior. Our goal is to identify the subset of program elements relevant to predicting the output for a given input. Considering only backward dependencies from the output may include statements that do not depend on the input, while considering only forward dependencies from the input may include computations that do not contribute to the output. To isolate the relevant computation, we focus on the intersection of these two views—i.e., *the program elements that both depend on the input and influence the output*. This *intersection* approximates the core input-to-output dependency path of the program and provides a principled basis for guiding attention toward more relevant tokens.

Let $I \subseteq V$ denote the *input seeds*, consisting of the formal parameters of each target method together with the actual arguments at all call sites that can reach a task sink. Let $O \subseteq V$ denote the *output seeds*, including the task sinks in S and any return or call-result nodes whose values flow into those sinks. In a simple `main`-style program, O includes the printed expression and any callee return that contributes to it. In a harness-based setting, O includes each assertion or equality predicate and any user-defined call results appearing within those predicates. Thus, O represents the set of observable values that determine the final answer.

To identify relevant program elements, we first compute the backward-reachable set from the output seeds,

$$R^-(O) = \{v \in V \mid \exists o \in O : \ v \rightsquigarrow o\}, \tag{2}$$

which captures nodes that may influence the output. However, this set may include statements that are independent of the input. We also compute the forward-reachable set from the input seeds,

$$R^+(I) = \{v \in V \mid \exists i \in I : \ i \rightsquigarrow v\}, \tag{3}$$

which captures nodes that depend on the input but may not affect the output. We then take the intersection of these two sets,

$$V_{\text{slice}} = R^+(I) \cap R^-(O), \tag{4}$$

to retain only the nodes that lie on dependency paths from inputs to outputs. This step filters out unrelated computations and yields a compact set of program elements that are both input-dependent and output-relevant. Finally, we include the governing control predicates of nodes in V_{slice} to ensure that branch conditions affecting the execution of output-relevant statements are preserved. We refer to the resulting set as a ***slice prior***: a slice-inspired relevance approximation designed to guide attention. This prior enables our steering mechanism to bias the model toward output prediction.

4.2 Token Weighting and Normalization

Our method adapts PASTA [25], an existing training-free, head-local attention-steering procedure to our problem. We replace user-highlighted text spans with a statically constructed relevance prior over code tokens, derived by static control flow analysis or program slicing, and we apply the steering update only to a sparse, automatically calibrated subset of heads during decoding [7]. Let $x_{1:n}$ denote the prompt tokens, consisting of the task instruction followed by a program, and let $y_{1:T}$ denote the generated answer. At decode step t, decoder layer ℓ, and attention head h, let

$$A^{(t)}_{\ell,h}(k) = \text{softmax}\Big(S^{(t)}_{\ell,h}\Big)_k, \tag{5}$$

be the attention probability assigned to valid key position $k \in \{1, \ldots, K_t\}$. Our goal is to bias part of this attention mass toward prompt positions that are more likely to be relevant to model reasoning on dynamic behaviors, while leaving model weights, tokenization, and the autoregressive decoding algorithm unchanged.

Each node $v \in V_{\text{slice}}$ (computed earlier) is mapped to its span in the source code and then to the prompt tokenization used by the model. Let $\Pi(v)$ denote the set of prompt-token indices aligned to node v. Before decode-time mixing, we assign zero mass to instruction tokens outside the code region and concentrate the prior on prompt tokens aligned to selected code nodes. This projection is tokenizer-dependent; when identifiers split into subword pieces, the induced prior may blur statement boundaries.

Each selected node receives a nonnegative weight $\omega(v)$ based on a set of conservative heuristics, including node type, proximity to the output seed, and whether the node lies on multiple input-to-output paths. These weights are accumulated on prompt tokens:

$$w_k = \sum_{v \in V_{\text{slice}} : k \in \Pi(v)} \omega(v) + \lambda \, \mathbf{1}[k \text{ belongs to a target-method body}], \tag{6}$$

where the optional baseline term $\lambda \geq 0$ prevents the prior from collapsing onto only a few tokens. The **static prompt prior** is

$$\bar{p}(k) = \frac{w_k}{\sum_{j=1}^{n} w_j}. \tag{7}$$

This prior is designed to encode likely semantic relevance, not capturing a complete dependence analysis for all program elements.

4.3 Decode-Time Prior and Steering Update

The static prior in Equation 7 is defined over prompt tokens, but generation unfolds over time and later decode steps also attend to previously generated tokens. To let the intervention evolve over generation without changing its basic form, we partition the continuation into equal-count temporal bins. The purpose of these bins is to let the prior emphasize different parts of the context at different phases of generation, for example, stronger prompt anchoring early in the explanation and more tolerance for generated context later.

Let $b(t)$ be the bin containing decode step t, let $\bar{p}^{(b)}$ denote the prompt prior used in bin b (in the simplest case, $\bar{p}^{(b)} = \bar{p}$ for all b), and let $r^{(t)}$ denote a recency prior over valid keys at step t. We define the decode-time prior as

$$p^{(t)} = \begin{cases} \bar{p}^{(b(t))}, & t = 1, \\ \text{Normalize}\Big((1 - \rho_{b(t)})\bar{p}^{(b(t))} + \rho_{b(t)} r^{(t)}\Big), & t > 1, \end{cases} \tag{8}$$

where $\rho_{b(t)} \in [0, 1]$ controls how strongly the current bin mixes prompt relevance with recency. For each steered layer ℓ and head h, we then reweight the already normalized attention distribution using this decode-time prior:

$$\widetilde{A}^{(t)}_{\ell,h}(k) = \frac{A^{(t)}_{\ell,h}(k) \left(p^{(t)}_k + \varepsilon\right)^{\beta}}{\sum_{j=1}^{K_t} A^{(t)}_{\ell,h}(j) \left(p^{(t)}_j + \varepsilon\right)^{\beta}}, \tag{9}$$

where $\beta > 0$ controls steering strength and $\varepsilon > 0$ is a small numerical-stability constant. Equation 9 redistributes existing attention mass toward positions with higher slice prior; it does not add external tokens or update model parameters.

4.4 Sparse Head Calibration

We do not steer every attention head. Instead, steering is restricted to a sparse subset of heads in each steered layer [21, 24]. This is both practical and methodological: the intervention should be strong enough to shift attention meaningfully, but narrow enough to avoid broad collateral disruption of unrelated attention patterns. We choose this subset automatically using a small calibration set C. For each layer-head pair (ℓ, h), we collect the attention distribution at the *first decode step* and score its alignment with the prompt prior:

$$g_{\ell,h} = \mathbb{E}_{x\sim C}\left[\sum_{k=1}^{n_x} A^{(1)}_{\ell,h}(k)\,\bar{p}_x(k)\right]. \tag{10}$$

We use the first decode step as it provides a stable prompt-conditioned attention profile before generated tokens begin to dominate the context. Heads with larger $g_{\ell,h}$ already attend more to slice-relevant prompt positions and are the most promising targets for steering.

Let $M_{\ell,h} \in \{0, 1\}$ denote the binary mask obtained by retaining only the highest-scoring fraction of heads in each steered layer. In the reported experiments, this fraction is held fixed within each model configuration and is roughly one quarter of the heads in each steered layer in our main setup; exact numerical values are deferred to Section 4. The final attention distribution is

$$\widehat{A}^{(t)}_{\ell,h} = A^{(t)}_{\ell,h} + M_{\ell,h}\left(\widetilde{A}^{(t)}_{\ell,h} - A^{(t)}_{\ell,h}\right). \tag{11}$$

If $M_{\ell,h} = 0$ for a head, or if $\beta = 0$, the steering operator reduces to the identity transformation for that head.

4.5 Decode-Time Steering Intervention

The intervention is applied only during autoregressive decoding, not during prompt prefill. This separation is important because our goal is to influence how the model *uses* the prompt while generating the explanation, rather than to alter the initial prompt encoding itself. In practice, we therefore build the KV cache from a prompt prefix without steering and begin generation from the final prompt token, so that the first steered step corresponds to a single-token query attending over the full prompt context.

To localize the intervention, we apply steering only in a late band of decoder layers, close to token selection. This provides enough influence near generation while keeping earlier representation formation largely untouched, thereby reducing collateral disruption to unrelated attention patterns and making the effect easier to attribute. Across all reported runs, the layer band, steering schedule, and steering strength are held fixed, while the active head subset is selected automatically by the calibration criterion in Equation 10.

5 Empirical Evaluation

We seek to answer the following research questions:

RQ1. Effectiveness of Model Steering. To what extent does our steering method recover LLM performance on obfuscated code?

RQ2. Sensitivity to Obfuscation Type. Which obfuscation techniques induce the most degradation in model performance, and which settings are most recoverable by steering?

RQ3. Behavioral Mechanisms of Obfuscation and Steering. How do obfuscation and steering alter both the reasoning trace as well as the model's internal attention dynamics during generation?

RQ3.1 Reasoning Trace-Level Behavior. How do obfuscation and steering change the quality and form of reasoning traces?

RQ3.2 Attention-Level Behavior. With obfuscated code, what head-level attention patterns distinguish reasoning-first from direct-answer outputs, and how are these patterns reshaped by steering?

RQ4. Generalization to Other Obfuscation Types. How well can CodeSteer be generalized to other obfuscation types?

6 Effectiveness of Model Steering (RQ1)

6.1 Output Prediction

6.1.1 Dataset, Procedure, and Metrics. For the output-prediction task, we reuse the same benchmark construction, prompting protocol, and case-level evaluation setup as in Section 3. We also used the same datasets as in Section 3: HumanEval [27] and CruxEval [10]. The programs in those two datasets allow us to isolate the effect of attention steering without confounding factors such as build failures, malware setup, missing dependencies, or unreliable harnesses. We chose our models as in Table 2 as they are strong open-weight code models with support for attention extraction and inference-time steering. Moreover, they were released before the chosen datasets, reducing the data-contamination risk. Each case is prompted and evaluated with three independent runs under the same decoding regime. The primary output-prediction metric remains **Pass@k**. To assess the effectiveness of CodeSteer, we define **Restoration Ratio** in attempt budget k as

$$R_P@k = \frac{P_{\text{Obfuscated+Steering}}@k - P_{\text{Obfuscated}}@k}{P_{\text{Original}}@k - P_{\text{Obfuscated}}@k} \times 100 \tag{12}$$

This measures the fraction of the obfuscation-induced performance drop that is recovered by steering. 100% means full recovery to the original code, values above 100% mean *over-recovery* (i.e., the steered obfuscated condition exceeds the original one), and negative values mean that steering hurts performance relative to the obfuscated one.

6.1.2 Empirical Result. As seen in Table 2, under steering, `Qwen2.5-7B` reaches 77.38–90.32% on HumanEval-X and 80.43%/ 89.14% / 91.83% on CruxEval-X; `DeepSeek-6.7B` reaches 74.89–89.11% and 70.25–89.63%. That is, even under obfuscation, CodeSteer allows all four models to maintain strong output-prediction performance, with the strongest absolute results obtained by `Qwen2.5-14B` and the weakest by `DeepSeek-6.7B`. As seen in Pass@1, steering improves first-attempt obfuscated performance in every setting. These Pass@1 numbers gain translates into a substantial recovery of the obfuscation-induced loss. At Pass@1, CodeSteer surpasses with 107.1% of the performance for `Qwen2.5-7B` on the original code in HumanEval-X, recovers 69.8% of the lost performance for `DeepSeek-6.7B`, 88.3% for `Qwen2.5-14B`, and 70.7% for `DeepSeek-V2-Lite`. On CruxEval-X, the restoration ratios are 77.1%, 27.6%, 81.1%, and 48.6%. That is, it not only improves obfuscated performance in absolute terms, but in many cases recovers a large fraction of the original-code gap, and in one setting (`Qwen2.5-7B`, HumanEval-X) even exceeds the original-code baseline.

Comparing across models, two complementary patterns emerge. First, stronger models remain stronger after steering: `Qwen2.5-14B` has the best absolute performance in every setting, followed by `DeepSeek-V2-Lite`, while `Qwen2.5-7B` and `DeepSeek-6.7B` remain lower in absolute accuracy. Second, the largest *relative* recovery does not always occur in the strongest model. In contrast, stronger models already preserve more useful semantics under obfuscation, so a

Table 2: Effectiveness of CodeSteer in output prediction (RQ1). Pass@k denotes case-coverage union across the first k runs for the same prompt and case tuple. Obfuscated conditions aggregate over the 4-obfuscation types. Values are percentages.

Model	Dataset	Original No-Obfuscation			Obfuscated w/o Prompt/Steer			Obfuscated w/. Prompt			Restoration Ratio w/ Prompt (%)			Obfuscated w/ Steering			Restoration Ratio w/ Steer (%)		
		P@1	P@2	P@3	P@1	P@2	P@3	P@1	P@2	P@3	R@1	R@2	R@3	P@1	P@2	P@3	R@1	R@2	R@3
Qwen2.5-7B	HumanEval	76.49	86.15	88.82	64.01	76.93	82.83	72.99	81.74	85.74	72.0	52.2	48.5	77.38	86.90	90.32	107.1	108.1	125.0
	CruxEval	83.11	92.90	95.34	71.40	83.16	87.81	79.91	89.68	95.37	72.7	66.9	100.4	80.43	89.14	91.83	77.1	61.4	53.4
DeepSeek-6.7B	HumanEval	77.74	84.84	88.40	68.29	81.43	86.47	65.91	84.52	90.08	-25.1	90.6	186.9	74.89	85.39	89.11	69.8	116.1	136.8
	CruxEval	78.98	88.96	91.98	66.93	81.63	88.44	70.36	82.75	91.83	28.5	15.3	95.8	70.25	84.43	89.63	27.6	38.2	33.6
Qwen2.5-14B	HumanEval	94.63	96.81	97.37	85.31	93.13	95.82	86.82	93.66	96.29	16.2	14.4	30.2	93.54	96.39	97.10	88.3	88.6	82.6
	CruxEval	92.14	97.28	98.33	86.48	94.56	96.95	89.42	96.25	98.15	51.9	62.1	87.1	91.07	96.70	98.17	81.1	78.7	88.4
DeepSeek-V2-Lite	HumanEval	90.31	94.78	96.08	80.18	90.62	92.37	68.00	82.35	88.19	-120.2	-198.8	-112.7	87.34	92.43	93.71	70.7	43.5	36.1
	CruxEval	91.56	95.87	97.12	81.11	91.39	94.66	79.66	93.79	97.42	-13.9	53.6	112.1	86.19	93.92	95.78	48.6	56.5	45.5

Table 3: CodeSteer's effectiveness on execution prediction

Model	Dataset	Orig.	Obf. no Pr or S	Obf. w/ Prompt	Restor. Ratio	Obf. w/ Steer	Restor. Ratio
		F1	F1	F1	R_{Pr}(%)	F1	R_{St}(%)
Qwen2.5-7B	HumanEval	36.52	33.12	27.82	-155.9	34.78	48.82
	CruxEval	43.35	39.32	38.49	-20.6	42.29	73.70
Qwen2.5-14B	HumanEval	40.14	38.17	36.31	-94.6	39.74	79.70
	CruxEval	49.20	45.88	45.6	-8.66	47.94	62.05

fixed steering configuration has less remaining error to correct, yielding smaller relative gains even when accuracy remains highest. Pass@2 and Pass@3 follow the same qualitative trend as Pass@1.

We also compared CodeSteer against the prompting approach in which we prompt a model under study to focus its attention on the set of collected code tokens as described in Section 4.2. As seen in Table 2, naive prompting is ineffective or less effective than CodeSteer. Pass@1 restoration ratios with prompting are lower in 7/8 rows or approximately the same in one row as CodeSteer.

To show its potential scalability, we chose the programs in DeepMind Code Contests dataset [15] with 300-600 LOCs (10×-20× longer than an average program in HumanEval-X and CRUXEval-X). Among all 820 programs, Joern was able to successfully perform slicing on 152 of them. Running Qwen2.5-7B on those programs, obfuscation reduced Pass@1 from 36.0% to 33.9%, and CodeSteer recovered it to 35.8%, yielding a 87.8% restoration rate. Model steering incurs no token cost. The maximum program size that CodeSteer can process is bounded by the underlying model's context length.

6.2 Execution-Trace Recovery

6.2.1 Dataset, Procedure, and Metrics. We use the same dataset as presented in Section 3, which contains 3,300 output-prediction cases in total. After removing 58 cases in HumanEval due to non-terminating execution, we obtain 3,242 validated execution-trace cases in total. We executed each program in the three settings (*original*, *obfuscated*, and *obfuscated code with Steering*) with its input and collected the execution trace. We compute **F1** against the oracle using longest-common-subsequence overlap. Let $L = \mathrm{LCS}(\hat{y}, y)$ be the longest common subsequence between the predicted trace $\hat{y}$ and the oracle trace y. We define $P_{\text{trace}} = \frac{L}{|\hat{y}|}, R_{\text{trace}} = \frac{L}{|y|}, F1_{\text{trace}} = \frac{2P_{\text{trace}}R_{\text{trace}}}{P_{\text{trace}}+R_{\text{trace}}}$. This gives partial credit when a model recovers the correct execution order partially. The F1 restoration ratio is defined as

$$R_{\text{F1}} = \frac{\text{F1}_{\text{trace(Obfuscated+Steering)}} - \text{F1}_{\text{trace(Obfuscated)}}}{\text{F1}_{\text{trace(Original)}} - \text{F1}_{\text{trace(Obfuscated)}}} \times 100 \qquad (13)$$

6.2.2 Empirical Result. Table 3 shows that steering improves the model performance on obfuscated code in every row of Table 3. For `Qwen2.5-14B`, F1 rises from 38.17 to 39.74 on HumanEval and from 45.88 to 47.94 on CruxEval, with restoration ratios of 79.70% and 62.05%. For `Qwen2.5-7B`, F1 rises from 33.12 to 34.78 on HumanEval and from 39.32 to 42.29 on CruxEval, corresponding to F1 restoration ratios of 48.82% and 73.70%. These results show that steering recovers part of the performance lost caused by obfuscation. Obfuscation also tends to add code constructs that expand the execution trace, adding more burden on a model to reason on obfuscated code.

In Table 2, Qwen models provide the strongest and most stable absolute performance in output prediction, with `Qwen2.5-14B` achieving the best results overall and `Qwen2.5-7B` remaining competitive under obfuscation. For Qwen, the larger model remains stronger in absolute F1, while the smaller model can still show substantial recovery under steering. Steering recovers part of the execution semantics lost under obfuscation, but the remaining difficulty is concentrated in reconstructing the entire ordered trace and values. Table 3 also shows that F1 scores with prompting on obfuscated code are lower or approximately the same as the no-prompt variant, thus largely ineffective as seen in the negative restoration ratios.

7 Sensitivity to Obfuscation Techniques (RQ2)

In this experiment, we study which obfuscation type degrades model performance the most and which is most recoverable by our steering. We reuses exactly the same output-prediction dataset, prompting procedure, and evaluation metrics as in Section 6.1. Table 4 shows that damage and recoverability vary substantially across obfuscation types. On HumanEval-X, the largest Pass@1 drop comes from *Identifier Renaming* (76.49→40.20), while *Control-Flow Flattening* is the least harmful (76.49→73.28). On CruxEval-X, the largest drop comes from *Dead-Code Injection* (83.11→66.08), whereas *Call Indirection* is the least harmful (83.11→77.16). Steering is most effective on *Control-Flow Flattening* for HumanEval-X (73.28→78.71, 169.10%) and on *Identifier Renaming* for CruxEval-X

Table 4: Stratified evaluation of Qwen2.5-7B model in output prediction (*pass*@1) over the four obfuscation types.

Dataset	Obfuscation Technique	Orig.	Obf. w/o Steering	Obf. w/ Steering	Restor. Ratio
H/Eval	Dead-Code Injection	76.49	70.54	76.79	105.00
	Call Indirection		72.00	75.73	83.06
	Control-Flow Flattening		73.28	78.71	169.10
	Identifier Renaming		40.20	78.30	104.99
C/Eval	Dead-Code Injection	83.11	66.08	77.15	64.99
	Call Indirection		77.16	83.01	98.28
	Control-Flow Flattening		67.84	76.08	53.99
	Identifier Renaming		74.51	85.48	127.50

(74.51→85.48, 127.50%). This confirms our intuition of *slice-based steering*. The weakest recoveries are *Call Indirection* on HumanEval-X (83.06%) and *Control-Flow Flattening* on CruxEval-X (53.99%).

This pattern matches our interpretation of obfuscation. Identifier renaming mainly disrupts lexical cues, so steering can recover performance by re-centering the model on relevant tokens rather than surface names. Dead-code injection and call indirection are less damaging to begin with, leaving less headroom for recovery.

8 Behavior Under Obfuscation and Steering

8.1 Model-Generated Reasoning Traces (RQ3.1)

While *pass@k* captures end-task success (as in Section 6), it does not reveal whether these arise from computationally light, reflexive responses (*System 1*) or from deliberate, structured reasoning (*System 2*). As noted in Section 1, obfuscation is expected to remove surface-level cues and suppress System 1 behavior, whereas steering aims to enforce System 2 reasoning. In this experiment, we examine how these factors influence generated reasoning traces.

8.1.1 The Quality of Generated Reasoning Traces. Following the restoration trends in Section 6, we select `Qwen2.5-7B` as the representative model for a fine-grained analysis of reasoning behavior. Among the obfuscation strategies, *identifier renaming* induces the largest degradation in performance (Table 4); therefore, to enable a more discriminative analysis, we focus on reasoning traces from this data subset. To construct a high-confidence reasoning pool, we only consider output prediction instances for which all three settings (*i.e.*, original, obfuscated, and steered) are answered correctly across all three trials, yielding a candidate subset of 452 instances across both datasets. Among these, we annotate six observable reasoning traits, grouped into three **positive** behaviors: (P1) *step-by-step execution reasoning*, (P2) *explicit state tracking or verification*, and (P3) *grounding in code statements*; and three **negative** behaviors: (N1) *surface-level lexical shortcuts*, (N2) *ungrounded or hallucinated reasoning steps*, and (N3) *direct answers without justification*.

To calibrate the annotation rubric, we manually inspect 20 samples and compare them against labels produced by Claude Opus 4.6, observing over 98.5% agreement; we then apply the same rubric to annotate an additional 150 samples, which are subsequently verified by human annotators. During labeling, the judge is provided with the original code and corresponding reasoning trace for the original setting, and with the obfuscated code and corresponding reasoning trace for both the obfuscated and steered settings.

Table 5: Pilot reasoning-behavior label rates for original-code reference artifacts, obfuscated plain artifacts, and obfuscated + steering artifacts. Values are percentages.

Reasoning Traits	Orig.	Obf. w/o Steering	Obf. w/ Steering
(P1) Step-by-step execution reasoning	74.2	71.3	87.3
(P2) Explicit state tracking or verification	58.9	58.7	70.7
(P3) Grounded to code statements	86.5	98.0	99.3
(N1) Surface-cue or lexical shortcut	6.3	10.7	2.7
(N2) Ungrounded leap or hallucinated step	28.7	24.0	28.7
(N3) Direct answer without reasoning	16.1	2.7	1.3

Table 5 summarizes the resulting label distributions across original programs, obfuscated variants, and obfuscated variants with steering, reporting the fraction of the labeled cases in which each reasoning trait appears in the rationale trace. Overall, *steering-guided* generation exhibits the highest prevalence of positive reasoning traits and the lowest incidence of negative ones, aligning with the relative performance trends observed across all program variants in output prediction (Table 2). Furthermore, these results support our broader hypothesis: obfuscation can shift the model away from reflexive System 1 lexical matching toward more System 2-style reasoning; however, the plain obfuscated condition still shows more surface-level cues or lexical shortcuts (10.7%) than either original code (6.3%) or steered code (2.7%). This suggests that *obfuscation alone does not reliably complete that shift*: when the code becomes confusing, the model can still *fall back on shallow priors unless steering keeps attention on semantically relevant statements*. Steering largely recovers performance toward the original model, but it also brings back some hallucination, as reflected by the relatively high rate of ungrounded leap or hallucinated step.

8.1.2 The Form of Generated Reasoning Traces. To assess the form of reasoning traces, we collect the following behavioral metrics:

(1) *Reasoning Present* (RP) indicates model engages in System 2–like behavior by producing a substantive explanation ($\geq$ 20 tokens),
(2) *Direct Answer* (DA) reflects System 1–like behavior, where the model outputs an answer with little or no reasoning,
(3) *Other/Unparsable* (O/U) captures low-quality outputs (e.g., malformed text) that do not link to coherent reasoning or answer,
(4) *#Tok* represents the average length of the reasoning trace ($\mu \pm \sigma$),
(5) *Avg-Stmt-Mentioned* is the average number of steering-guided statements explicitly mentioned in the reasoning traces,
(6) *Stmt-Recall* is the fraction of steered statements in the traces.

In this experiment, we use the stored per-run generations from `Qwen2.5-7B` under all three evaluation settings.

In Table 6, for the *correctly predicted instances*, we can see a clear shift toward *System 2*–like reasoning, as illustrated by the increasing RP (65.0→70.8→76.3) and subsequent decreasing DA (27.7→21.9→15.1). In these cases, reasoning length also increases (117→135→144), indicating deeper reasoning processes. Moreover, the number of statements in the program slice that are explicitly mentioned in the traces is the highest with steering (on an average, 4.36 per program). From Table 6, the statement recall is 26.2%. Thus, with 4.36 referenced statements, the model explicitly makes references for every (4.36*100)/26.2 = 16.7 statements on average in

Table 6: Trace-level behavioral and grounding metrics. "RP"=reasoning present, "DA"=direct answer, "O/U"=other, and "Tok" = avg. rationale tokens among reasoning-present runs (reported as mean±std). The last two columns measure steered-element grounding among reasoning-present rationales. Percentages are computed within the displayed correctness split. Model=`Qwen2.5-Coder-7B`. Corr: Correct prediction.

Orig.	Obf.	Steer.	Corr. Pred.	RP (%)	DA (%)	O/U (%)	#Tok	Avg-Stmt-Mentioned	Stmt-Recall
✓	–	–	y	65.0	27.7	7.3	117 ± 50	2.737	16.5%
–	✓	–	y	70.8	21.9	7.3	135 ± 74	2.519	19.3%
–	✓	✓	y	76.3	15.1	8.6	144 ± 89	**4.361**	**26.2%**
✓	–	–	n	43.2	27.5	29.3	157 ± 93	3.091	15.1%
–	✓	–	n	44.9	24.6	30.5	179 ± 102	2.796	17.3%
–	✓	✓	n	52.2	20.5	27.3	186 ± 104	3.279	18.9%

the verbalized reasoning trace toward predicting the correct output. The strong alignment between higher RP and improved grounding (*Avg-Stmt-Mentioned* and *Stmt-Recall*) suggests ***successful predictions occur when reasoning is present and is well grounded***.

For *incorrectly predicted instances*, RP drops markedly (by 50.5%–57.7.%–46.2%) alongside a sharp increase in O/U and trace length, with reasoning traces also becoming less grounded (lower Avg-Stmt-Mentioned and Stmt-Recall). This implies incorrect predictions are associated with misdirected or incoherent System 2 reasoning. Key behavioral differences between correct and incorrect runs are: (1) correct predictions consistently exhibit a higher proportion of RP (65.0–76.3%) than incorrect ones (43.2–52.2%), indicating that successful predictions are strongly associated with coherent System 2 reasoning; (2) incorrect runs show a much higher O/U rate, suggesting instability or breakdown in reasoning generation; and (3) incorrect runs tend to have longer yet ineffective rationales. This highlights that ***reasoning length alone does not guarantee correctness***, and reasoning quality and coherence are critical.

Overall, these findings reveal that ***obfuscation reduces reliance on System 1 behavior, while steering promotes System 2 reasoning***; but only coherent and well-directed reasoning leads to correct predictions. Accordingly, steering improves both the *frequency* of reasoning (RP) and its *quality* (*i.e.*, grounding), but only for correct predictions. Importantly, this improvement does not arise from simply eliciting more verbose rationales; rather, steering guides the model to focus on code elements most relevant to the task outcome. This underscores that effective reasoning requires not just engaging System 2 thinking, but is grounded in the right elements.

8.2 Attention Patterns (RQ3.2)

In this experiment, we investigate the model behavioral changes at the attention level in two transitions: (1) from running on the original code to the obfuscated code, and (2) from the obfuscated code to the running on the steered model. Specifically, we analyzed the head-level attention features from `Qwen2.5-Coder-7B-Instruct` during our output prediction experiment under three conditions: original code, obfuscated code, and obfuscated code with steered model. These features are collected as part of the experiment in RQ1, and are stored as a per-head summary tensor for each run. For each attention head, this tensor records the attention mass allocated to *code tokens*, *instruction tokens*, and *other tokens* in the prompt, as well as other statistics such as attention entropy and head output norm. We then compare these head-level features in the above two transitions. We report all features separately for the sets of *correct* and *incorrect* instances for output prediction.

In particular, we present *Cohen's d effect size* computed from a per-run *code-bias score*. For a given attention head h and run n, we define the run-level *code-bias score* as

$$\text{CodeBias}_{h,n} = \texttt{attn\_mass\_code}_{h,n} - \texttt{attn\_mass\_instr}_{h,n}, \qquad (14)$$

where $\texttt{attn\_mass\_code}_{h,n}$ is the ***total attention mass assigned by head h in run n to code-specific tokens***, and $\texttt{attn\_mass\_instr}_{h,n}$ is the ***total attention mass assigned to natural-language instruction tokens***. Intuitively, $\text{CodeBias}_{h,n}$ *measures whether that head is more focused on the code itself or on the surrounding instruction text in that run*. Then, we compute these run-level scores within each group under comparison. Specifically, for groups A and B with N_A and N_B runs, we define

$$\mu_{A,h} = \frac{1}{N_A} \sum_{n \in A} \text{CodeBias}_{h,n}, \qquad \mu_{B,h} = \frac{1}{N_B} \sum_{n \in B} \text{CodeBias}_{h,n},$$

and let $\sigma_{A,h}$ and $\sigma_{B,h}$ denote the corresponding standard deviations of the run-level CodeBias scores within each group. We then compute the pooled standard deviation

$$s_{p,h} = \sqrt{\frac{(N_A - 1)\sigma_{A,h}^2 + (N_B - 1)\sigma_{B,h}^2}{N_A + N_B - 2}}$$

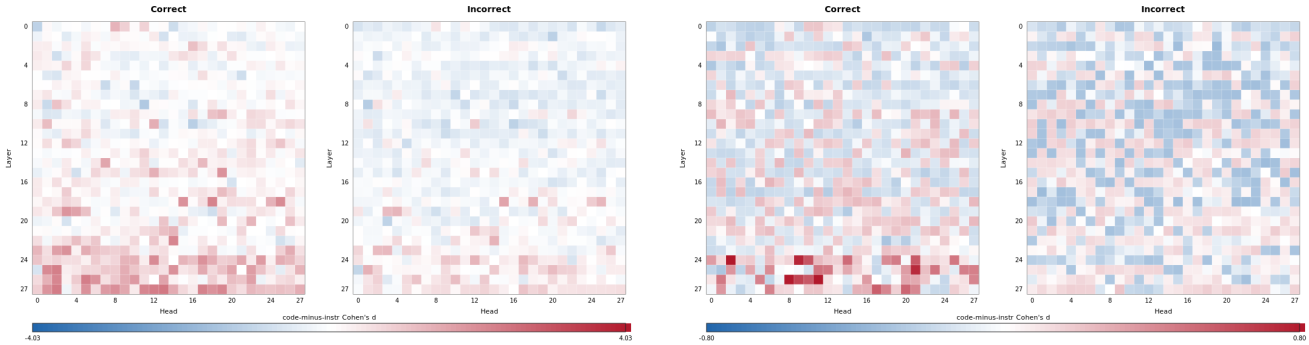
and define the head-level Cohen's d score as $d_h = \frac{\mu_{A,h} - \mu_{B,h}}{s_{p,h}}$.

Cohen's d is a standardized effect size. Positive values mean that the first-named condition in the comparison is relatively more code-focused at that head, whereas negative values mean that it is relatively more instruction-focused.

As shown in Figure 3(a), for the correctly predicted cases, the last four layers contain a noticeably higher density of dark red heads. This indicates that, under obfuscation, the model's late layers allocate substantially more attention to code tokens than to other tokens. In contrast, this pattern is largely absent in the incorrect cases. This observation is consistent with our finding in RQ2.1: obfuscation shifts the model toward stronger focus on code elements, encouraging a more statement-by-statement reasoning style analogous to System 2 reasoning in humans.

We next examine the change from the non-steered obfuscated runs to the steered runs in Figure 3(b). For the correct cases, the overall code-focused trend remains, but steering refines this broad shift into a more selective head-level pattern, with some already focused red heads becoming even darker. That is, our steering-guided inference does not simply make all late-layer heads attend more strongly to code. Instead, it concentrates stronger code-focused attention into a smaller subset of late-layer heads, while allowing other heads to remain more sensitive to instruction tokens.

By contrast, the incorrect cases appear more diffuse and less selectively organized. Their pattern is spread more evenly across layers and heads, with much weaker concentration on a specific subset of late-layer heads than in the correct cases. This suggests that successful reasoning is associated not only with stronger attention to code, but also with a selective allocation of that attention.



(a) Obfuscated run *VS* Non-Obfuscated run **(b) Steered obfuscated run *VS* Non-Steered Obfuscated run**

Figure 3: Attention-level behavioral changes. Each heatmap reports a head-level code-minus-instruction Cohen's d map for the correct and incorrect sets under one pairwise contrast. Positive values indicate the relatively more code-focus at that head.

9 Other Obfuscation Type Generalization (RQ4)

Empirical Setting. To evaluate CodeSteer's generalization potential, we chose four new types of obfuscation. First, in the ***opaque-predicate*** obfuscation, a deterministic guard is added to preserve the original execution path while introducing a syntactically plausible decoy branch. For example, a direct statement such as `realCode();` can be wrapped as `if ((x*x+x)%2==0) realCode(); else decoy();`. This transformation tests whether the model can ignore an irrelevant but plausible branch without treating the decoy as the semantic target. Second, the ***loop transformation*** obfuscation preserves the iteration order but changes the loop structure. For example, a `for` loop is rewritten as a `while` loop with separated initialization and update logic. Third, the ***branch inversion*** obfuscation negates the predicate and swaps the branch bodies, e.g., rewriting `if (c) A else B` as `if (!c) B else A`. This transformation preserves control-flow semantics but requires the model to track predicate polarity correctly. Fourth, the ***arithmetic/boolean rewriting*** replaces direct expressions with equivalent algebraic or bitwise forms. For example, `a+b` can be rewritten as `(a^b)+2*(a&b)`. This preserves the computed value but forces a model to recognize semantic equivalence.

From each dataset (HumanEval-X and CruxEval-X), we randomly sampled 10 eligible snippets for each of the above obfuscation types, ensuring that each obfuscation type can be applied to the selected code, yielding 80 additional code instances with 320 input-output prediction cases. Each transformed variant is validated against the original oracle so that the evaluated behavior remains unchanged. We used `Qwen2.5-Coder-7B-Instruct` as in the experiment in Section 7 and recorded the evaluation metrics for the original, obfuscated, and obfuscated-with-CodeSteer settings.

Empirical Result. As seen in Table 7, CodeSteer improves opaque-predicate cases from 63.75% to 71.25% with a restoration ratio of 75%. This result shows that the static slice prior does not pre-determine which branch the model should take, yet helps it focus on the output-deciding branches, steering the model from drifting toward the decoy branch. Interestingly, CodeSteer makes the model perform even better than itself on the original code (74.58% vs 73.75%). Loop transformation spreads the same loop-carried state across different ways of initialization, guard, and update statements. This shows that our static prior does not presolve the loop iteration, yet it helps a model keep its reasoning about statements in the loop structurally connected to the queried output. Table 7 also shows that static priors do not help for arithmetic/boolean rewriting (67.19% vs. 67.08%) and branch inversion (71.07% vs. 70.83%). These obfuscation types mainly change the semantic meaning of a single expression or predicate. Thus, the static priors across statements do not help.

A case study. Figure 4 shows a loop-transformation case study (others can be found in project's page [1]). The transformation splits the loop state across a list-building loop and a consuming loop (`while` to `for` at line 8). As seen in the reasoning traces: without steering, the model correctly follows the helper return value but loses the task mapping from helper return to the assertion predicate, while steering keeps the loop-state reasoning connected to the final assertion being evaluated, leading to correct output prediction.

Table 7: Output prediction effectiveness on additional obfuscation types; values are P@1 aggregate accuracies (%); Obf. w/ Steering: steered model on obfuscated code (RQ4)

Obfuscation Technique	Orig.	Obf. w/o Steering	Obf. w/ Steering	Restor. Ratio
Opaque Predicates	73.75	63.75	71.25	75.00
Loop Transformation		67.50	74.58	113.28
Arithmetic/Boolean Rewriting		67.19	67.08	-1.68
Branch Inversion		71.07	70.83	-8.96

Input. Case 1: `assert(f("flak") == false)`. Case 2: `assert(f("aa") == false)`.

```
1  List<Character> ls = new ArrayList<>();
2  int position = 0;
3  while (position < w.length()) {
4    ls.add(w.charAt(position));
5    position++;
6  }
7  StringBuilder omw = new StringBuilder();
8  for (; !ls.isEmpty();) {
9    omw.append(ls.remove(0));
10   if (ls.size()*2 > w.length() &&
11       w.substring(ls.size()).equals(omw.toString()))
12     return true;
13 }
14 return false;
```

Baseline: "For the input 'flak', this condition is not met during any iteration... Since the condition is not met and the list is emptied, the function returns 'false'."
CodeSteer: "The input string is 'flak'... the condition `ls.size() * 2 > w.length()` is never met, and the function returns false."

Figure 4: Loop-transformation reasoning trace example

Obfuscated C vulnerability-trace case study

Concrete input: AAAAAAAAAAAAAAAAAAAAAAAA (24 characters).

```c
#include <stdio.h>
#define X1 16
#define X2 128
int z9(const char *q) {
    char a[X1];
    int b = 0, c = 0, d = 0;
    for (;;) {
        if (!(q[d] != '\0')) break;
        char e = q[d];
        b += (e >= '0' && e <= '9') ? (e - '0') :
             (e >= 'A' && e <= 'Z') ? 2 :
             ((e ^ e) + 1);

        *(a + d) = e;

        if ((d & 1) && !(d & 1)) {
            b += 999;
        }
        d++;
    }
    *(a + d) = '\0';
    c = ((b ^ 21) < (b ^ 1)) ? 1 : (b > 20);
    int (*p)(const char*, ...) = printf;
    (c ? p : p)(c ? "High score.\n" : "Low score.\n");
    p("Copied string: %s\n", a);
    p("Score: %d\n", b);
    return c;
}

z9((const char *)(y + 0));
```

Output from the LLM without steering

"The input string is treated as having length 28, and the local buffer `a` in `z9` is treated as having capacity 16. The loop continues copying characters into `a[d]` until `d` reaches 28, so the input is considered much larger than the destination buffer. **Execution trace:** [30, 31, 32, 33, 5, 6, (7, 8, 9, 10-12, 14, 16, 19) x16, 7, 8, 9, 10-12, 14] The program may crash or behave erratically before producing reliable later output. Execution stops at the write on 'L019' when 'd = 16', because that is treated as the first buffer-overflow point.
Output: `Enter text:` The program may crash or behave erratically before producing reliable later output."

Output from the LLM with CodeSteer

"The `z9` function processes the input character by character until the null terminator is encountered. The local buffer `a` has size 16, but the concrete input is 24 characters long. When the index reaches 16, the next character write goes past the end of `a`, and after the loop the terminating NUL write also lands out of bounds. The score accumulator still reaches 48, so the score branch is `High score.`, and the print sink is reached even though the copied string itself is no longer reliable after memory corruption. Execution trace: [30, 31, 32, 33, 5, 6, (7, 8, 9, 10-12, 14, 16, 19) x24, 7, 8, 21, 22, 23, 24, 25, 26, 27, 35] Conclusion: The input length is 24, the destination buffer capacity is 16, the sink is reached, the character copy writes past the end of the buffer, the terminating null byte also writes past the end of the buffer, and the overall result is a buffer overflow.

Figure 5: Buffer-overflow vulnerability analysis

10 Case Study on Vulnerability Analysis

In this section, we present a case study on how we used CodeSteer to perform vulnerability analysis on an obfuscated C program with buffer overflow. In Figure 5, the code was obfuscated with alpha-renaming. The local buffer `a` has capacity 16, yet the input has 24 characters. We used `Qwen3-Coder-30B-A3B-Instruct` in two settings: regular inference and slice-guided steering. Without steering, the model recognizes the presence of a memory-safety issue, but its reasoning diverges from the concrete execution. Specifically, it overestimates the input length from 24 to 28 characters and retreats to a generic crash explanation rather than maintaining the precise vulnerability-relevant execution trace in the program. However, via static analysis, the slice prior emphasizes the vulnerability-relevant dataflow around the local buffer `a`, the loop index `d`, the copy write `*(a + d) = e`, the terminating null write `*(a + d) = '\0'`, and the relevant statements needed to determine if execution reaches these out-of-bound writes. With steering, the model preserves the correct input length and buffer capacity, traces the copy loop to the first out-of-bounds write, and correctly identifies the final NUL terminator write as a second out-of-bounds access.

11 Related Work

Code Obfuscation and Program Comprehension [5, 6]. Nguyen *et al.* [17] reported that obfuscated code reduces humans' accuracy in output prediction and induces a cognitive shift from memory-based reasoning (System 1) to slower, analytical reasoning (System 2).

LLMs for Code Understanding. LLMs have achieved strong performance across a wide range of code-related tasks [2, 9, 11, 18, 22]. We show that LLMs often rely on surface-level patterns, e.g., identifier names and common idioms, rather than deep semantic reasoning.

LLMs under Obfuscation. Nikiema *et al.* [19] showed that performance degrades significantly as obfuscation complexity increases. Similarly, Le *et al.* [13] studied how removing meaningful naming and structural cues affects LLM reasoning. PoorCodeSumEval [12] evaluates model robustness under degraded code readability, including obfuscation-like transformations. Fang *et al.* [8] examined LLMs' effectiveness in code analysis tasks such as deobfuscation.

Attention Steering in LLMs. In PASTA [25] and AutoPASTA [26], reweighting attention toward selected tokens can improve model faithfulness and reasoning performance. However, these approaches are largely task-agnostic and do not leverage program semantics.

Static Analysis. Static analysis aims to estimate program properties over all possible inputs. [3, 4] In contrast, we position CodeSteer as a data-driven *static emulation* that is *complementary*. Rather than seeking input-independent guarantees via engineered abstract domains, it predicts the most likely *concrete, per-input* execution.

12 Threats to Validity and Conclusion

Internal validity. Threats include errors in the obfuscation pipeline, slice construction, and steering implementation. Results may depend on prompts, calibration choices, and hyperparameters.

External validity. Our study is limited to a small set of LLMs, and benchmark tasks focused on output and execution prediction. The findings may not fully generalize to other languages, obfuscation types, security workflows, despite the promising case study in C.

Construct validity. Our tasks do not cover all aspects of program comprehension. Our slice is an approximation of semantic relevance rather than a fully sound one, so the results are viewed as evidence of improved task reasoning rather than full code understanding.

CodeSteer combines program analysis with attention steering to recover the performance that LLMs lose under obfuscation in various tasks. Our behavioral analysis suggests that steering helps shift the model away from System 1's lexical shortcuts and toward more System 2, code-grounded stepwise reasoning under obfuscation.

13 Data Availability Statement

All code and data is available at [1].

References

[1] Anonymous. 2026. CodeSteer. doi:10.5281/zenodo.19340781

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[3] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting Decompiler Output with Learned Variable Names and Types. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4327–4343. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin

[4] Mihai Christodorescu and Somesh Jha. 2003. Static Analysis of Executables to Detect Malicious Patterns. In *12th USENIX Security Symposium (USENIX Security 03)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/12th-usenix-security-symposium/static-analysis-executables-detect-malicious-patterns

[5] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A Taxonomy of Obfuscating Transformations. *Technical Report* (1997).

[6] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 184–196. doi:10.1145/268946.268962

[7] Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. Towards Automated Circuit Discovery for Mechanistic Interpretability. arXiv:2304.14997 [cs.LG] https://arxiv.org/abs/2304.14997

[8] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. 2024. Large Language Models for Code Analysis: Do LLMs Really Do Their Job? arXiv:2310.12357 [cs.SE] https://arxiv.org/abs/2310.12357

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[10] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. CRUXEval: a benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 659, 54 pages.

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[12] Chao Hu, Yitian Chai, Hao Zhou, Fandong Meng, Jie Zhou, and Xiaodong Gu. 2024. How Effectively Do Code Language Models Understand Poor-Readability Code?. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 795–806. doi:10.1145/3691620.3695072

[13] Cuong Chi Le, Minh V. T. Pham, Cuong Duc Van, Hoang N. Phan, Huy N. Phan, and Tien N. Nguyen. 2025. When Names Disappear: Revealing What LLMs Actually Understand About Code. arXiv:2510.03178 [cs.SE] https://arxiv.org/abs/2510.03178

[14] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. doi:10.1126/science.abq1158

[15] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (Dec. 2022), 1092–1097. doi:10.1126/science.abq1158

[16] Tímea László and Ákos Kiss. 2007. Obfuscating C++ Programs via Control Flow Flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica* 30.

[17] Anh H. N. Nguyen, Jack Le, Ilse Lahnstein Coronado, and Tien N. Nguyen. 2026. The Effect of Code Obfuscation on Human Program Comprehension. arXiv:2603.07668 [cs.SE] https://arxiv.org/abs/2603.07668

[18] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG] https://arxiv.org/abs/2203.13474

[19] Serge Lionel Nikiema, Jordan Samhi, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F. Bissyandé. 2025. The Code Barrier: What LLMs Actually Understand? arXiv:2504.10557 [cs.SE] https://arxiv.org/abs/2504.10557

[20] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security and Privacy* 9, 5 (2011), 41–47. doi:10.1109/MSP.2011.98

[21] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2022. In-context Learning and Induction Heads. arXiv:2209.11895 [cs.LG] https://arxiv.org/abs/2209.11895

[22] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 3473–3487. doi:10.18653/v1/2024.emnlp-main.203

[23] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2019. An Observational Investigation of Reverse Engineers' Process and Mental Models. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems* (Glasgow, Scotland Uk) *(CHI EA '19)*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3290607.3313040

[24] Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. 2022. Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small. arXiv:2211.00593 [cs.LG] https://arxiv.org/abs/2211.00593

[25] Qingru Zhang, Chandan Singh, Liyuan Liu, Xiaodong Liu, Bin Yu, Jianfeng Gao, and Tuo Zhao. 2024. Tell Your Model Where to Attend: Post-hoc Attention Steering for LLMs. arXiv:2311.02262 [cs.CL] https://arxiv.org/abs/2311.02262

[26] Qingru Zhang, Xiaodong Yu, Chandan Singh, Xiaodong Liu, Liyuan Liu, Jianfeng Gao, Tuo Zhao, Dan Roth, and Hao Cheng. 2024. Model Tells Itself Where to Attend: Faithfulness Meets Automatic Attention Steering. arXiv:2409.10790 [cs.CL] https://arxiv.org/abs/2409.10790

[27] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Long Beach, CA, USA) *(KDD '23)*. Association for Computing Machinery, New York, NY, USA, 5673–5684. doi:10.1145/3580305.3599790

Received 2026-03-26; accepted 2026-06-18