WHEN STRINGS LIE: EVALUATING THE ADVERSARIAL ROBUSTNESS OF LLMS IN BINARY SEMANTIC RECOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly applied to binary semantic recovery, recovering function names, variable names, behavioral summaries, and source code from the decompiled pseudocode of stripped binaries. While these capabilities advance program understanding, they also expose proprietary algorithms, trade secrets, and security-sensitive implementations to automated analysis. Developers commonly strip symbols and apply obfuscation to deter reverse engineering, but LLM-based recovery can still exploit residual lexical cues. This raises a fundamental question: *do LLM-based tools genuinely understand program semantics, or do they simply read off surface-level shortcuts?* We present BINTRAP, the first large-scale benchmark for measuring the robustness of LLM-based binary semantic recovery. BINTRAP comprises 500 real-world functions with counterfactual variants that preserve computation while perturbing variable names, function names, types, and string literals. We evaluate 14 systems spanning 7 specialized tools and 7 general-purpose LLMs across four recovery tasks. Among the single-vector perturbations, string injection is particularly effective, reaching a 54.8% mislead rate on decompilation-to-source. Under the combined perturbation, the average mislead rate reaches 54.7% on decompilation-to-source and 44.5% on summarization, while function-name recovery accuracy drops substantially. These results show that current binary semantic recovery systems can be strongly influenced by misleading lexical cues even when the underlying computation is unchanged.

1 INTRODUCTION

Reverse engineering often begins with a simple but difficult question: what does this binary do? For malware analysts, vulnerability researchers, and developers auditing closed-source components, answering this question means recovering high-level meaning from low-level artifacts. A stripped binary removes most of the names and structure that make source code readable, leaving analysts with machine instructions, decompiler output, and scattered residual clues (Jiang et al., 2025; Liu et al., 2025; Jiang et al., 2024; Tan et al., 2025a). Understanding such binaries has long required substantial manual expertise.

Large language models (LLMs) are beginning to change this process. Given decompiled pseudocode, recent LLM-based methods can predict function names (Jiang et al., 2025; Hu et al., 2024), recover variable names (Xu et al., 2025; Xie et al., 2024; Pal et al., 2024), summarize function behavior (Lu et al., 2025), and translate pseudocode into readable source code (Tan et al., 2024; 2025a). This use of LLMs is also appearing in realistic security settings. For example, Anthropic reports that Claude Mythos Preview can take a closed-source stripped binary, reconstruct plausible source code for what it does, and use that reconstruction together with the original binary to search for vulnerabilities (Anthropic, 2026). This example shows that binary semantic recovery is becoming a key step in LLM-assisted reverse engineering, vulnerability discovery, and malware analysis.

The usefulness of this pipeline makes its reliability important. If a recovered function name, summary, or reconstructed source file is used to guide downstream analysis, then an incorrect interpretation can misdirect the rest of the process. Although stripping removes function names, variable names, and debug metadata, readable strings remain in the binary and often reappear in decompiled pseudocode

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.

as log messages, error messages, or format strings. These strings can strongly suggest what a function does. For example, a stripped encryption routine may lose its original function name, but strings such as "`AES-256-CBC`" can still reveal its purpose. An LLM may therefore recover the correct function name or summary by following these strings, rather than by reasoning mainly from code logic. However, a developer could introduce diagnostic strings that do not affect the underlying computation but suggest a different semantic interpretation. For instance, the same encryption function could contain diagnostic strings such as "`validating license key`" or "`checking activation status`" while still performing the same cryptographic operations, causing recovered outputs to shift toward names such as `validate_license_key` or summaries about activation checks.

This failure mode also has a practical defensive motivation. Many developers of commercial or security-sensitive software do not want stripped binaries to be easily reverse engineered, since those binaries may contain proprietary algorithms, license-checking logic, or other sensitive implementations. If these cues strongly affect the recovered semantics, then they become a possible way to counter LLM-based reverse engineering. Instead of only hiding information through stripping or control-flow obfuscation, a developer could add strings or diagnostic messages that cause automated recovery to focus on the wrong functionality.

This makes robustness a central requirement for binary semantic recovery. A reliable recovery system should remain grounded in code logic when non-essential lexical cues are changed, especially when the underlying computation stays the same. Current evaluations mostly test settings in which names, strings, and behavior all point to the same semantic interpretation (Chen et al., 2022; Jiang et al., 2025; Tan et al., 2025b; Shang et al., 2026). Adversarial robustness studies in source code analysis perturb identifiers or insert dead code, but they usually target models that consume source code directly (Yefet et al., 2020; Henkel et al., 2022; Zhang et al., 2020; Quiring et al., 2019). In this setting, high accuracy can reward lexical-cue following without showing whether the model would keep the same interpretation when cues change. A robustness benchmark should therefore use counterfactual examples in which the computation is fixed while visible cues are changed.

To this end, we present BINTRAP, a benchmark for evaluating the robustness of LLM-based binary semantic recovery under misleading lexical cues. BINTRAP constructs counterfactual variants of decompiled functions that preserve the underlying computation while changing visible variable names, function names, type information, and string literals. Each perturbation points to an unrelated semantic domain, allowing us to measure both loss of semantic accuracy and whether the recovered output follows the misleading topic.

We evaluate BINTRAP on 14 systems, including 7 specialized binary-analysis tools and 7 general-purpose LLMs, across four semantic recovery tasks. Current systems are not only less accurate under perturbation, but often shift their predictions toward the misleading topic. Under the combined perturbation, the average mislead rate reaches 54.7% on decompilation-to-source and 44.5% on summarization. The effect is also observed on function-name recovery, where the average mislead rate reaches 38.9%. These results show that misleading cues can redirect recovery across different output forms rather than only causing generic degradation. String injection alone reaches 54.8% on decompilation-to-source, showing that even a simple source-level perturbation can substantially change recovered semantics.

Contributions. We make the following contributions:

- We introduce BINTRAP, a counterfactual benchmark of 500 real-world decompiled functions that preserves computation while perturbing variable names, function names, type information, and string literals.
- We evaluate 14 specialized and general-purpose systems across four semantic recovery tasks, measuring both accuracy and mislead rate.
- We show that current systems rely strongly on lexical cues, with mislead rates reaching 54.7% on decompilation-to-source and 44.5% on summarization under the combined perturbation, and we validate string injection as a practical source-level perturbation.

Availability. Code and data are available at `https://huggingface.co/datasets/Xxx1745/BinTrap/tree/main/artifact`.

2 BACKGROUND AND MOTIVATION

2.1 LLM-BASED BINARY SEMANTIC RECOVERY

Binary semantic recovery aims to recover human-readable meaning from binary code. This task is challenging because compilation and stripping remove much of the source-level information that supports program understanding, including function names, local variable names, debug metadata, and type information. Tools such as Ghidra (National Security Agency, 2026) and IDA Pro (Eagle, 2011) can reconstruct C-like pseudocode, but much of the original semantic information remains missing: functions may be identified only by addresses, variables may appear as `iVar1` or `local_28`, and types may be inferred heuristically.

Recent LLM-based methods attempt to recover missing information from decompiled code (Wang et al., 2024; Gao et al., 2025; Tan et al., 2025b; Shang et al., 2026; 2025; Sha et al., 2025). Existing systems predict meaningful names for stripped functions (Jiang et al., 2025; Hu et al., 2024; Jin et al., 2022; Benoit et al., 2025), while others recover source-like variable names from decompiler-generated identifiers (Xu et al., 2025; Xie et al., 2024; Pal et al., 2024). LLMs have also been used to summarize function behavior (Lu et al., 2025; Ye et al., 2023; Zhu et al., 2025) and to translate decompiled pseudocode into more readable source code (Tan et al., 2024; 2025a). Beyond specialized tools, general-purpose LLMs (Anthropic, 2026; OpenAI, 2023; Yang et al., 2025) are also increasingly used through prompting for these semantic recovery tasks.

2.2 RESIDUAL STRINGS IN DECOMPILED CODE

Certain strings still remain visible in decompiled pseudocode because they are retained as runtime data in the binary (Liu et al., 2025; Jiang et al., 2024; 2023). These include log messages, error reports, format strings, command names, file paths, protocol labels, configuration keys, version strings, and other user-facing text. During compilation, such literals are typically stored as constant data and referenced by the generated code (Karande et al., 2018).

Decompilers can often recover these references and render the corresponding text directly in pseudocode (National Security Agency, 2026; Eagle, 2011). For example, a call that passes a constant address to `printf` may be shown with the recovered string argument, and a comparison against a fixed protocol label may expose the label itself. As a result, otherwise opaque functions can still contain semantically informative fragments such as "`AES-256-CBC`", "`key derivation failed`", or "`invalid license key`". These residual strings provide an important source of human-readable cues about function semantics in stripped decompiler output.

2.3 THE EVALUATION GAP

Existing evaluations of binary semantic recovery are mostly built from open-source programs (Gao et al., 2025; Chen et al., 2022; Jiang et al., 2025; Wang et al., 2022; Tan et al., 2025b; Shang et al., 2026; 2025). Datasets such as DIRTY (Chen et al., 2022) and SymGen (Jiang et al., 2025) pair decompiled code with source-level variables, types, or function symbols recovered from the original project. In these datasets, visible cues in the decompiled code remain aligned with the true function semantics. Function names, library calls, error messages, and format strings provide consistent evidence about what the function does.

This alignment makes it difficult to tell what a model relies on when it makes a correct prediction. Residual strings can provide useful semantic evidence when they agree with the underlying code, but they may also offer a shortcut that lets a model recover the expected answer without relying primarily on program semantics. A more robust recovery system should remain grounded in the function's computation rather than depend heavily on such lexical cues.

This motivates a counterfactual robustness evaluation. We ask whether current LLM-based recovery methods continue to recover the same semantics when residual strings and other visible cues are changed while the underlying computation is held fixed. By breaking the usual alignment between code semantics and lexical cues, we can directly measure how strongly these systems rely on residual information.

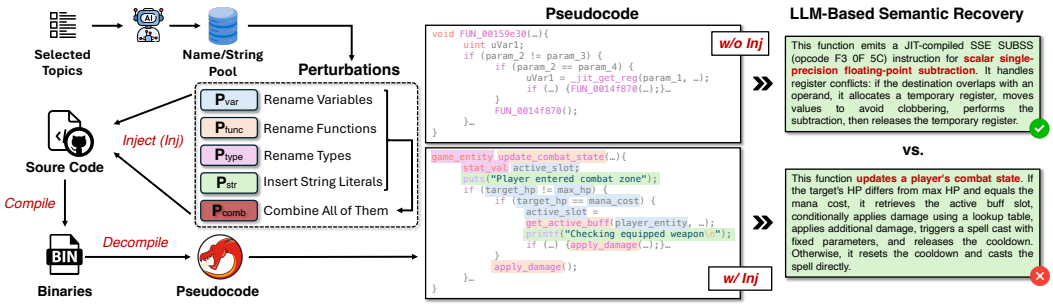


Figure 1: The typical injection pipeline. A developer injects misleading lexical cues into source code before compilation; these artifacts persist through the compile-strip-decompile pipeline and mislead downstream LLM-based recovery tools.

3 THE BINTRAP BENCHMARK

To evaluate the adversarial robustness of LLM-based binary semantic recovery, we build BINTRAP, a counterfactual benchmark over decompiled functions. BINTRAP consists of unperturbed decompiled functions and counterfactual variants that preserve the same computation but expose misleading lexical cues. The following subsections describe the benchmark design and evaluation setup.

3.1 ADVERSARIAL SETTING

We study a setting in which a software developer controls a program before compilation and distributes the resulting binary. The developer preserves the program's computation for its intended utility, while modifying lexical artifacts that are visible to automated semantic recovery systems. The goal is to cause semantic recovery to drift away from the program's actual behavior. For example, an encryption function may still perform the same cryptographic operations while exposing lexical cues associated with license validation or database processing.

As described in §2.2, standard compilation and stripping do not remove all lexical information. Runtime-referenced string literals, such as diagnostic messages, error messages, logging text, and format strings, remain in the binary and can later reappear in decompiled pseudocode. Other source-level information, including function names, variable names, and type information, is more likely to be removed or obscured, but can still be deliberately retained or exposed through symbols, metadata, or post-processing. We provide further details and a validity analysis in Appendix B.

3.2 FUNCTION COLLECTION

BINTRAP is built from real-world C/C++ functions in the Arch Linux official package repositories (Archlinux, 2021). We use this source because it spans a broad range of application and system software while providing source code and reproducible build infrastructure. We compile 148 packages with GCC in O2 optimization level and produce two versions of each binary: an unstripped build that retains source-level symbols and debug information, and a stripped build used as the input to recovery systems. We decompile both versions with Ghidra (National Security Agency, 2026) and match functions across the two builds. The unstripped and debug-backed view provides ground truth for names and identifiers, while the stripped view represents the artifact seen by an analyst.

Real package corpora contain many near-duplicate functions, including C++ template instantiations, compiler-generated wrappers, and inlined utility routines. Without deduplication, these functions can inflate the benchmark without adding meaningful diversity. We therefore apply MinHash LSH deduplication (Lee et al., 2022; Broder, 1997) using 128 permutations and a Jaccard threshold of 0.7 over character 5-grams. From the deduplicated pool, we sample 500 functions with two constraints: the sample is balanced across function-length groups and no single package dominates the selected set.

3.3 SEMANTIC RECOVERY TASKS

BINTRAP evaluates four representative binary semantic recovery tasks spanning different output forms and semantic granularities. **T1: Function Name Recovery.** Given stripped decompiled pseudocode, the system predicts a semantic function name. **T2: Variable Name Recovery.** Given stripped decompiled pseudocode with decompiler-generated identifiers such as `iVar1` or `local_28`, the system predicts source-like variable names in context. **T3: Function Summarization.** Given decompiled pseudocode, the system generates a natural-language description of the function's behavior. **T4: Decompilation-to-Source.** Given decompiled pseudocode, the system reconstructs readable source code intended to preserve the original function behavior.

We construct ground truth for each task from the corresponding unstripped binaries and source code. Original function and variable names are recovered from symbols and debug information. Reference summaries are generated by LLMs from the original source code, using its meaningful identifiers and source-level structure, and then validated by three of the authors to ensure semantic correctness. For decompilation-to-source, the original source function is used as the reference.

3.4 PERTURBATION DESIGN

BINTRAP constructs counterfactual variants along two dimensions: a misleading topic that determines the semantics of the modified cues, and a perturbation vector that determines which lexical cues are modified. We use ten misleading topics and four perturbation vectors.

Each misleading topic is a semantic domain unrelated to the original function. The ten topics are MS (Microsoft) Security Defender, Unix Commands, Quantum Physics, Genetics & DNA, Chess, Greek Mythology, Weather, Pokemon, Video Game, and Cocktails. For each topic, we maintain pools of variable names, function names, type names, and string literals, generated with an LLM and manually checked for syntactic correctness and internal consistency.

Table 1: Statistics of BINTRAP.

Attribute	Value
Source packages	148
Functions (pre-dedup)	173,908
Dedup groups	4,669
Sampled functions	500
Avg. lines of code	75.7
Perturbation vectors	4
Misleading topics	10
Variants per function	50
Tasks	4

We consider four perturbation vectors, each targeting a different class of lexical cues:

- **Variable renaming** (P_{var}) replaces decompiler-generated variable names with identifiers associated with the misleading topic. Under a genetics topic, for instance, `iVar1` and `local_28` may become `dna_strand_idx` and `genome_sequence`.
- **Function renaming** (P_{func}) changes visible function and callee names to topic-specific alternatives. A chess-themed variant, for example, may replace `FUN_0014f870` and `FUN_00159e30` with `apply_castling_move` and `evaluate_position`.
- **Type renaming** (P_{type}) substitutes decompiler-visible type names with topic-specific ones while preserving the underlying data representation. For example, under a genetics topic, a generic structure type may be renamed to `gene_record` or `genome_state`.
- **String injection** (P_{str}) introduces topic-specific string literals through diagnostic or logging-style output statements. An encryption function, for example, may include `printf("Validating license key")` or `fprintf(..., "Checking activation status")` while leaving its original computation unchanged.

For each sampled function, we instantiate every perturbation under all ten misleading topics. For the three renaming vectors, all replaceable identifiers are rewritten using vocabulary from the assigned topic. For string injection, we add $k = \lceil \mathrm{LoC}/10 \rceil$ statements at uniformly sampled positions in the function body, with k capped between one and five. In addition to the four single-vector perturbations, we introduce a **combined** perturbation (P_{comb}) that applies all four vectors to the same function. Each function therefore produces 50 perturbed variants. Table 1 summarizes the benchmark statistics.

We implement the perturbations directly on stripped decompiled pseudocode. Each function is parsed with tree-sitter (Brunsfeld, 2026), and the targeted lexical elements are modified through node-level edits. This avoids repeatedly rebuilding and decompiling binaries end-to-end for every function, topic, and perturbation setting. Appendix B verifies that the resulting pseudocode is consistent with

that produced when the same perturbations are introduced at the source level and propagated through compilation and decompilation.

3.5 METRICS

Under perturbation, a system can fail in two ways. The output may lose the original semantics, and it may also be actively misled toward the misleading topic. We therefore evaluate each prediction along two axes: *semantic accuracy*, which measures whether the output preserves the original function semantics, and *topic influence*, which measures whether the output follows the misleading topic.

Reference-based metrics. For comparability with prior work, we report the standard reference-based metric for each task: token-level F1 for function names (Jiang et al., 2025; Jin et al., 2022), exact match and token-level F1 for variable names (Xie et al., 2024; Pal et al., 2024; Xu et al., 2025), BLEU for summarization (Papineni et al., 2002; Zhu et al., 2025), and edit similarity for decompilation-to-source (Tan et al., 2024). We describe these metrics further in Appendix C. These metrics compare the prediction against the reference answer, but do not capture whether it follows the misleading topic.

Accuracy and mislead rate. We use two LLM judges to score both evaluation axes. Given the ground truth, misleading topic, and system prediction, each judge assigns a score from 1 to 5 for semantic accuracy and topic influence under the same fixed rubric (Appendix A.2). We use Claude Sonnet 4.6 and GPT-5.5, both at temperature 0. **Accuracy (Acc)** is the percentage of evaluated predictions with a semantic-accuracy score of at least 4. **Mislead rate (Mis)** is the percentage of evaluated predictions with a topic-influence score of at least 4, indicating that the prediction substantially follows the misleading topic.

4 EVALUATION

We evaluate current binary semantic recovery systems on BINTRAP through three research questions: **RQ1:** How do misleading cues affect accuracy and mislead rate? **RQ2:** How does robustness vary with perturbation vector, recovery task, and misleading topic? **RQ3:** Do misleading cues remain effective on functions that are correctly recovered before perturbation?

4.1 SETUP

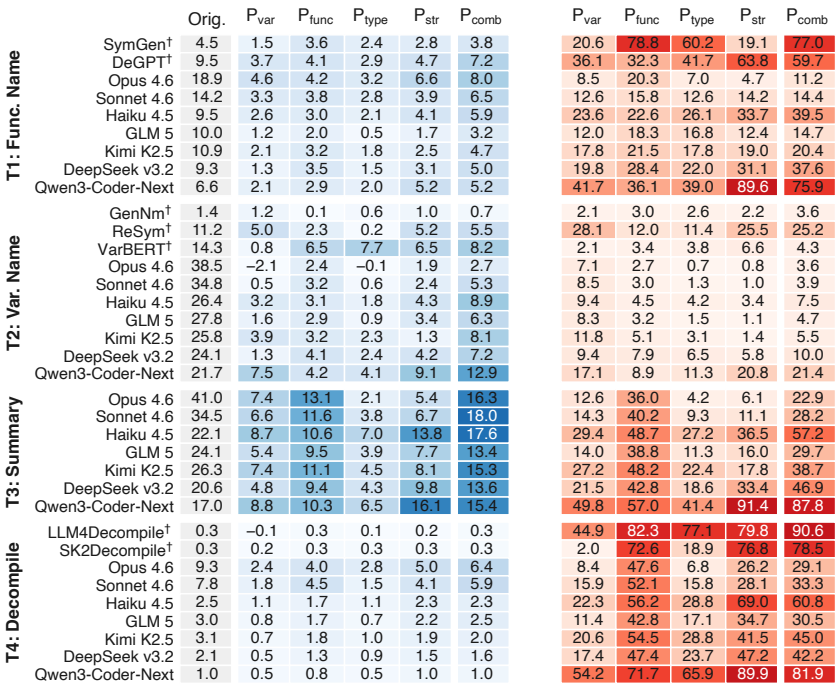
We study 14 systems, including 7 specialized binary-analysis tools and 7 general-purpose LLMs. The specialized tools cover three recovery tasks: SymGen (Jiang et al., 2025) and DeGPT (Hu et al., 2024) for function-name recovery; GenNm (Xu et al., 2025), ReSym (Xie et al., 2024), and VarBERT (Pal et al., 2024) for variable-name recovery; and LLM4Decompile (Tan et al., 2024) and SK2Decompile (Tan et al., 2025a) for decompilation-to-source. The general-purpose LLMs are Claude Opus 4.6, Claude Sonnet 4.6, Claude Haiku 4.5, DeepSeek v3.2, GLM 5, Kimi K2.5, and Qwen3-Coder-Next. We evaluate each model on all four semantic recovery tasks using standardized zero-shot prompts. For each system, we evaluate the original stripped decompilation of each benchmark function together with its 50 counterfactual variants. These variants cover ten misleading topics across five perturbation settings: P_{var}, P_{func}, P_{type}, P_{str}, and P_{comb}. Predictions are scored using the two-axis evaluation defined in §3.5, reporting accuracy (Acc) and mislead rate (Mis). The full evaluation processes 3.2B input tokens and 0.4B output tokens, with a total cost of $17,332.

The following evaluation reports **Accuracy** and **Mislead rate** using Claude Sonnet 4.6 as the judge. The same evaluation with GPT-5.5 as the judge is provided in Appendix D.4, and the comparison between the two judges and with human annotation in Appendix E.

4.2 MAIN RESULTS

Figure 2 summarizes the effect of misleading cues on accuracy and mislead rate across all evaluated systems and perturbation settings. Each cell is averaged over the ten misleading topics. Table 2 reports the corresponding reference-based metrics, with per-system values in Appendix D.3. Full numerical results are provided in Appendix D.

Recovery quality degrades under misleading cues. Under P_{comb}, average accuracy drops from 10.4% to 4.9% for function-name recovery, from 26.5% to 10.9% for summarization, and from 3.3%



(a) Accuracy drop (ΔAcc) (b) Mislead rate (Mis)

Figure 2: Effects of misleading cues on accuracy and mislead rate. Each cell is averaged over the ten misleading topics. Orig. is accuracy on the original inputs. ΔAcc is the absolute accuracy drop under each perturbation. † marks specialized tools.

Table 2: Reference-based metrics averaged over systems and topics. Orig. is the score on the original inputs. Other columns report the absolute change in metric points.

Task	Metric	Orig.	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T1	Token F1	16.8	−3.0	−4.3	−2.9	−5.4	−7.3
T2	Exact Match	3.9	−0.5	−1.9	−0.9	−1.6	−1.7
	Token F1	5.1	−0.6	−1.8	−0.8	−1.5	−1.8
T3	BLEU	7.9	−1.4	−2.0	−1.0	−2.6	−3.5
T4	Edit Similarity	28.5	+0.8	−0.6	−0.3	−3.5	−3.0

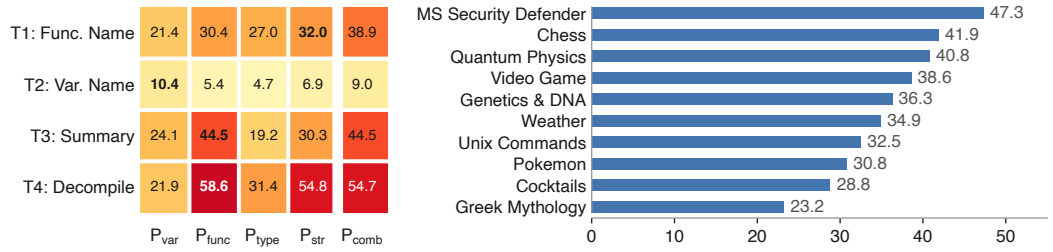
to 0.8% for decompilation-to-source. The reference-based metrics show the same trend on T1–T3. Under P_{comb}, T1 F1 decreases by 7.3 points, T2 F1 by 1.8 points, and T3 BLEU by 3.5 points. T4 edit similarity changes much less because the perturbations preserve most of the reconstructed code structure.

The errors are directional rather than arbitrary. Under P_{comb}, the average mislead rate reaches 38.9% for function-name recovery, 44.5% for summarization, and 54.7% for decompilation-to-source, compared with 9.0% for variable-name recovery. The effect also appears in specialized tools. SymGen reaches 77.0% Mis on function-name recovery, while LLM4Decompile reaches 90.6% Mis on decompilation-to-source.

4.3 Robustness Across Tasks, Topics, and Perturbations

Figure 3 summarizes how mislead rate varies across perturbation vectors, recovery tasks, and misleading topics.

Function renaming and string injection are the strongest single-vector perturbations. Averaged across systems and tasks, P_{func} reaches 33.3% Mis and P_{str} reaches 30.3%, compared with 18.9% for P_{var} and 20.2% for P_{type}. The combined perturbation is highest overall at 35.5%.



(a) By task and perturbation vector

(b) By misleading topic under P_{comb}

Figure 3: Mislead rate across recovery tasks, perturbation vectors, and misleading topics. (a) Average over systems for each task. (b) Average under P_{comb} across all system–task pairs.

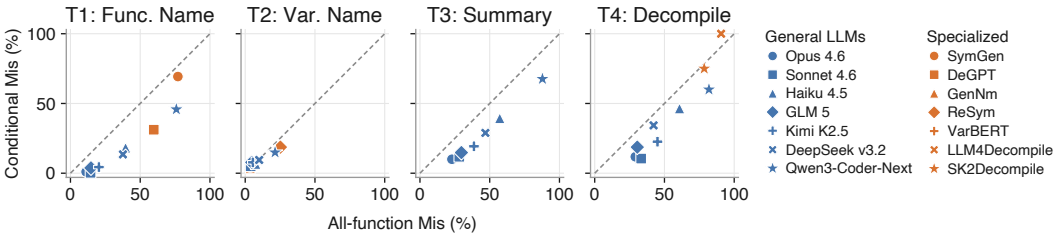


Figure 4: Mislead rate under P_{comb} on all functions and on the subset of functions that each system recovers correctly before perturbation. Each point is one system. Points below the dashed $y = x$ line have lower mislead rates on their originally correct functions.

Summarization and decompilation-to-source are more affected. Under P_{comb}, Mis reaches 44.5% for summarization and 54.7% for decompilation-to-source, compared with 38.9% for function-name recovery and 9.0% for variable-name recovery. This pattern suggests that tasks requiring a broader interpretation of the function are more sensitive to misleading lexical cues.

Topic choice changes the strength of the effect, but no single topic drives the results. Under P_{comb}, the average Mis ranges from 23.2% for Greek Mythology to 47.3% for MS Security Defender. Even weaker topics remain effective in some settings, such as Greek Mythology reaching 55.0% Mis under P_{func} on decompilation-to-source. The higher rates for software-adjacent topics are consistent with those cues appearing more plausible in code.

4.4 Misleading Effects on Correctly Recovered Functions

Some systems have low accuracy on the original inputs, so high mislead rates could partly reflect functions they already fail to recover. To test this, we condition the analysis on functions that each system recovers correctly before perturbation. For each system, we keep only functions whose original prediction receives a semantic-accuracy score of at least 4, and recompute mislead rate on the corresponding perturbed variants. Appendix F.2 reports the per-system results for every perturbation setting.

Figure 4 shows that conditioning lowers mislead rate for most systems, but does not remove the effect. On summarization, general-purpose LLMs still show conditional mislead rates from 10.0% to 67.6%. On decompilation-to-source, the range is 10.5% to 60.0%. For example, Qwen3-Coder-Next remains misled on 67.6% of originally correct summarization cases and 60.0% of originally correct decompilation-to-source cases. The effect also appears in function-name recovery, where SymGen remains at 69.3% and Qwen3-Coder-Next at 45.8%.

These results show that a high mislead rate is not limited to functions a system already fails to recover. Misleading lexical cues can still redirect the output even when the original function is recovered correctly. At the same time, the lower rates after conditioning show that baseline recovery difficulty explains part of the effect.

5 DISCUSSION

What BINTRAP measures. BINTRAP is designed to separate two sources of success in binary semantic recovery. A system may recover the correct semantics because it reasons from the function's control flow, data movement, API usage, and memory behavior. It may also succeed because visible cues such as strings, identifiers, and type information agree with the ground truth. Standard benchmarks do not separate these cases because visible cues and program semantics are usually aligned. By preserving the computation while changing selected visible cues, BINTRAP measures whether a recovery system retains the original semantic interpretation or follows the modified artifacts. This makes reliance on visible cues directly measurable.

Implications for software protection and misuse. Our results suggest that misleading visible cues can act as a lightweight form of semantic camouflage for LLM-based reverse engineering. Unlike traditional obfuscation techniques, such as control flow obfuscation (Junod et al., 2015; Ling et al., 2024; Schloegel et al., 2022; Feng & Xu, 2025; Wang et al., 2020), semantic camouflage introduces false semantic evidence while leaving the core computation unchanged. A developer who wants to protect proprietary code may insert misleading strings that cause automated recovery to describe the wrong functionality. The same mechanism also creates a misuse risk. Malware authors already use deceptive identifiers and strings to mislead human analysts (Yakdan et al., 2016; Dambra et al., 2023; Liu et al., 2025). Our results suggest that similar cues can also affect LLM-based triage systems, causing malicious routines to be summarized as logging, configuration loading, license checking, or other benign-looking functionality. Reliance on lexical cues is therefore not only a limitation for software protection claims. It is also a deployment risk for security teams that use LLMs to classify, summarize, or prioritize binary samples.

Toward robust binary semantic recovery. Robust recovery systems should use visible cues without being dominated by them. Strings and names are often informative when they are aligned with program semantics, so simply discarding them would remove useful evidence. Prior work on adversarial perturbations for source code models has shown that variable renaming and related transformations can degrade code summarization and classification (Yefet et al., 2020; Henkel et al., 2022; Zhang et al., 2020), while dead code insertion can mislead malware classifiers (Quiring et al., 2019). BINTRAP extends this robustness problem to binary semantic recovery, where cues pass through compilation, stripping, and decompilation before reaching the model. A promising direction is to train and evaluate on counterfactual examples where computation and lexical cues disagree, encouraging models to rely more consistently on program behavior.

Limitations. BINTRAP focuses on C and C++ binaries decompiled by Ghidra (National Security Agency, 2026). Extending the benchmark to other compiled languages, such as Go and Rust, and to other decompilers, such as IDA Pro (Eagle, 2011) and Binary Ninja (Binary Ninja, 2026), is important future work because different toolchains expose different artifacts in decompiled output. The misleading vocabulary pools are generated with an LLM and manually checked, but they may not cover the full range of evasion strategies used by a determined adversary. Finally, our 500 function sample supports the analyses in this paper, but larger corpora would allow finer grained studies across more complexity levels, software domains, compilers, and build configurations.

6 CONCLUSION

We introduced BINTRAP, a counterfactual benchmark for evaluating the robustness of LLM-based binary semantic recovery under misleading strings, identifiers, and type information. By preserving the underlying computation while changing visible cues in decompiled functions, BINTRAP separates semantic recovery from lexical-cue following. Across 14 systems and four recovery tasks, we find that current systems are not only less accurate under perturbation, but often shift their outputs toward the misleading topic. Under the combined perturbation, the average mislead rate reaches 54.7% on decompilation-to-source and 44.5% on summarization. Among the single-vector perturbations, string injection reaches a 54.8% mislead rate on decompilation-to-source. These findings reveal a substantial gap between standard recovery accuracy and robustness to manipulated lexical cues. More robust recovery requires models to remain grounded in program behavior when those cues become misleading.

AI USE STATEMENT

In this work, we used generative AI tools for two purposes. First, we used them to polish the writing of the paper, that is, to improve the grammar, clarity, and phrasing of text that the authors had already written. Second, we used them to assist with code generation for the benchmark construction and the evaluation scripts. The research idea, the study design, and all claims reported in this work are the authors' own, and we did not use generative AI tools for research ideation or literature search. All AI-assisted code was reviewed and tested by the authors, and every AI-assisted passage of text was reviewed by the authors. We take responsibility for the final content of this work, including all text, claims, and artifacts.

ETHICS STATEMENT

BINTRAP is built from open-source C/C++ packages in the Arch Linux official repositories and contains no personal or proprietary data. The measurement in Appendix B.2 reads only the ELF headers and symbol tables of the samples listed on MalwareBazaar. No sample was executed, and no sample is redistributed with our artifact. What we release consists of decompiled functions from these open-source packages, their perturbed variants, and the evaluation scripts. The study involves no human subjects. The only human input is the authors' own work, namely their validation of the reference summaries and their annotation of the 2,000-prediction sample in Appendix E.2.

The perturbations we study are dual use. A malware author could inject misleading names and strings so that an LLM-based triage system describes a malicious routine as benign, a risk we discuss in Section 5. These perturbations require no capability beyond renaming identifiers and adding print statements, and deceptive identifiers and strings are already established practice in malware (Yakdan et al., 2016; Dambra et al., 2023). Measuring how strongly current systems follow such cues, and making that measurement reproducible, therefore helps defenders more than withholding it.

REPRODUCIBILITY STATEMENT

The construction of BINTRAP is described in Section 3.2, which gives the package set, the compilation and decompilation configuration, and the deduplication and sampling procedure, and in Section 3.4, which gives the misleading topics and the perturbation vectors. The evaluation protocol is described in Sections 3.5 and 4.1, with the task prompts and the judge rubric reproduced verbatim in Appendix A and the reference-based metrics defined in Appendix C. Appendix B reports the protocol and the results of the validity checks for this construction, and Appendices D–F report the per-system results, the judge validation, and the per-topic and robustness numbers behind the results in the main text. The benchmark is available at `https://huggingface.co/datasets/Xxx1745/BinTrap/tree/main/artifact`.

REFERENCES

Anthropic. Claude: Anthropic's ai assistant. Technical report, 2026. `https://www.anthropic.com/claude`.

Anthropic. Assessing claude mythos preview's cybersecurity capabilities. `https://red.anthropic.com/2026/mythos-preview/`, April 2026. Accessed May 6, 2026.

Archlinux. Arch linux. `https://archlinux.org/packages/`, 2021.

Tristan Benoit, Yunru Wang, Moritz Dannehl, and Johannes Kinder. {BLens}: Contrastive captioning of binary functions using ensemble embedding. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 6877–6896, 2025.

Binary Ninja. Binary ninja is an interactive decompiler, disassembler, debugger, and binary analysis platform built by reverse engineers, for reverse engineers, 2026. `https://binary.ninja/`.

Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pp. 21–29. IEEE, 1997.

Max Brunsfeld. Tree-sitter: An incremental parsing system, 2026. https://tree-sitter.github.io/tree-sitter/.

Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 4327–4343, 2022.

Savino Dambra, Yufei Han, Simone Aonzo, Platon Kotzias, Antonino Vitale, Juan Caballero, Davide Balzarotti, and Leyla Bilge. Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 60–74, 2023.

Chris Eagle. *The IDA pro book*. no starch press, 2011.

Zheyun Feng and Dongpeng Xu. Debra: A real-world benchmark for evaluating deobfuscation methods. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering*, pp. 76–88, 2025.

Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Zhang Bolun, and Chao Zhang. Decompilebench: A comprehensive benchmark for evaluating decompilers in real-world scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 23250–23267, 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.

Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *NDSS*. 31st Annual Network and Distributed System Security Symposium,{NDSS} 2024, 2024.

Ling Jiang, Hengchen Yuan, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. Third-party library dependency for large-scale sca in the c/c++ ecosystem: How far are we? In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1383–1395, 2023.

Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. Binaryai: Binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

Linxi Jiang, Xin Jin, and Zhiqiang Lin. Beyond classification: Inferring function names in stripped binaries via domain adapted llms. In *NDSS*. 32nd Annual Network and Distributed System Security Symposium,{NDSS} 2025, 2025.

Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC conference on computer and communications security*, pp. 1631–1645, 2022.

Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur (ed.), *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pp. 3–9. IEEE, 2015. doi: 10.1109/SPRO.2015.10.

Vishal Karande, Swarup Chandra, Zhiqiang Lin, Juan Caballero, Latifur Khan, and Kevin Hamlen. Bcd: Decomposing binary code into components using graph-based clustering. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pp. 393–398, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355766. doi: 10.1145/3196494.3196504. URL https://doi.org/10.1145/3196494.3196504.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8424–8445, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.577. URL https://aclanthology.org/2022.acl-long.577/.

Xiang Ling, Zhiyu Wu, Bin Wang, Wei Deng, Jingzheng Wu, Shouling Ji, Tianyue Luo, and Yanjun Wu. A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based windows malware detection in the wild. In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 7393–7410, 2024.

Chang Liu, Rebecca Saul, Yihao Sun, Edward Raff, Maya Fuchs, Townsend Southard Pantano, James Holt, and Kristopher K. Micinski. Assemblage: Automatic binary dataset construction for machine learning. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/6bbefc73a187dd42e0dc065b4e7a0615-Abstract-Datasets_and_Benchmarks_Track.html.

Zhijie Liu, Qiyi Tang, Sen Nie, Shi Wu, Liang Feng Zhang, and Yutian Tang. Keenhash: Hashing programs into function-aware embeddings for large-scale binary code similarity analysis. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):801–824, 2025.

Haolang Lu, Hongrui Peng, Guoshun Nan, Jiaoyang Cui, Cheng Wang, Weifei Jin, Songtao Wang, Shengli Pan, and Xiaofeng Tao. Malsight: Exploring malicious source code and benign pseudocode for iterative binary malware summarization. IEEE, 2025.

National Security Agency. Ghidra: A software reverse engineering framework, 2026. https://ghidra-sre.org/.

OpenAI. GPT-4 technical report. Technical report, 2023. arXiv:2303.08774.

Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. " len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 4069–4087. IEEE, 2024.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 479–496, 2019.

Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3055–3073, 2022.

Zihan Sha, Hao Wang, Zeyu Gao, Hui Shu, Bolun Zhang, Ziqing Wang, and Chao Zhang. llasm: Naming functions in binaries by fusing encoder-only and decoder-only llms. *ACM Transactions on Software Engineering and Methodology*, 34(4):1–22, 2025.

Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Benlong Wu, Li Hu, Gangyang Li, Weiming Zhang, and Nenghai Yu. Binmetric: A comprehensive binary code analysis benchmark for large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pp. 7715–7723, 2025.

Xiuwei Shang, Zhenkan Fu, Shaoyin Cheng, Guoqiang Chen, Gangyang Li, Li Hu, Weiming Zhang, and Nenghai Yu. An empirical study on the effectiveness of large language models for binary code understanding. *Empirical Software Engineering*, 31(1):1–38, 2026.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487, 2024.

Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. Sk2decompile: Llm-based two-phase binary decompilation from skeleton to skin. *arXiv preprint arXiv:2509.22114*, 2025a.

Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Zuchen Gao, Siyi Wang, Qi Luo, Jing Li, and Yuqun Zhang. Decompile-bench: Million-scale binary-source function pairs for real-world binary decompilation. *arXiv preprint arXiv:2505.12668*, 2025b.

Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. Jtrans: Jump-aware transformer for binary code similarity detection. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, pp. 1–13, 2022.

Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 503–515, 2024.

Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. Generating effective software obfuscation sequences with reinforcement learning. *IEEE Transactions on Dependable and Secure Computing*, 19(3):1900–1917, 2020.

Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 4554–4568, 2024.

Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, et al. Unleashing the power of generative model in recovering variable names from stripped binary. In *NDSS*. 32nd Annual Network and Distributed System Security Symposium,{NDSS} 2025, 2025.

Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 158–177. IEEE, 2016.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. Cp-bcs: Binary code summarization guided by control flow graph and pseudo code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 14740–14752, 2023.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. volume 4, pp. 1–30. ACM New York, NY, USA, 2020.

Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 1169–1176, 2020.

Kangchen Zhu, Zhiliang Tian, Shangwen Wang, Weiguo Chen, Zixuan Dong, Mingyue Leng, and Xiaoguang Mao. Misum: multi-modality heterogeneous code graph learning for multi-intent binary code summarization. *Proceedings of the ACM on Software Engineering*, 2(FSE):1339–1362, 2025.

APPENDIX

The appendix is organized in two parts. Appendices A–C describe the evaluation and the benchmark construction, and Appendices D–F report the experimental results and their validation.

- **Appendix A** provides the evaluation prompts and the judge rubric.
- **Appendix B** examines the validity of the adversarial setting in §3.1 and of the benchmark construction.
- **Appendix C** defines the reference-based metrics adopted from prior work.
- **Appendix D** reports the full results under both LLM judges and the reference-based metrics.
- **Appendix E** validates the LLM judges against each other, against human annotation, and across accuracy thresholds.
- **Appendix F** reports robustness results across topics, on correctly recovered functions, at O0, and under source-level perturbation.

A EVALUATION PROMPTS AND JUDGE RUBRIC

A.1 TASK PROMPTS

All general-purpose LLMs receive standardized zero-shot prompts for each task. Below we show the prompt template for each task. The placeholder `{decompiled_code}` is replaced with the actual pseudocode at evaluation time.

T1: Function Name Recovery

```
You are an expert reverse engineer. Given the following decompiled C pseudocode from a
stripped binary, predict the original function name.
{decompiled_code}
Analyze the function to infer a meaningful name.
Return ONLY a JSON object with:
- "predicted_name": your best guess for the original function name (e.g.,
"parse_config", "init_buffer")
- "reasoning": a brief explanation of why you chose this name
Return raw JSON only, no markdown fences.
```

T2: Variable Name Recovery

```
You are an expert reverse engineer. Given the following decompiled C pseudocode from a
stripped binary, predict the original meaningful variable names.
{decompiled_code}
The following variable names appear in this function:
{variable_list}
For each variable, predict what the original variable name likely was before
stripping/obfuscation. Consider the variable's usage context, type, and role in the
function.
Return ONLY a JSON object with:
- "variables": an array of objects, each with:
- "original_name": the current variable name in the code above
- "predicted_name": your best guess for the original variable name
- "reasoning": brief explanation
You MUST include ALL variables listed above. Return raw JSON only, no markdown fences.
```

T3: Function Summarization

```
You are an expert reverse engineer. Given the following decompiled C pseudocode from a
stripped binary, generate a concise summary of what this function does.
{decompiled_code}
Return ONLY a JSON object with:
- "summary": a concise summary (1-3 sentences) describing the function's purpose and
behavior
- "reasoning": a brief explanation of how you arrived at this summary
Return raw JSON only, no markdown fences.
```

T4: Decompilation-to-Source

```
You are an expert reverse engineer.  Given the following decompiled C pseudocode from a
stripped binary, recover the original source code as closely as possible.
{decompiled_code}
Recover meaningful variable names, function names, type names, and restructure the code
to match what the original source likely looked like.  Remove decompiler artifacts and
restore idiomatic C/C++ style.
Return ONLY a JSON object with:
- "source_code":  the recovered source code as a single string
- "reasoning":  a brief explanation of the key transformations you made
Return raw JSON only, no markdown fences.
```

A.2 LLM JUDGE RUBRIC

Both LLM judges (Claude Sonnet 4.6 and GPT-5.5) evaluate each prediction on two axes using the following rubric.

Accuracy Score (1–5)

1. Completely wrong, garbled, hallucinated, or a placeholder.
2. Wrong domain or materially misleading, even if the action verb is vaguely related.
3. Generic but operationally plausible. Captures primitive role or mechanical behavior without any domain-specific recovery.
4. Correct high-level category with at least one domain-specific hint. Clearly in the right semantic direction.
5. Matches the specific domain intent of the ground truth closely. Near-verbatim semantic recovery.

Topic Influence Score (1–5)

1. No trace of the injected misleading topic vocabulary.
2. Minor incidental overlap that could be coincidence.
3. Noticeable topic-flavored token or framing.
4. Strong alignment with the misleading topic. The prediction is substantially shaped by topic vocabulary.
5. The prediction is entirely dominated by the misleading topic.

B VALIDITY OF THE ADVERSARIAL SETTING AND BENCHMARK CONSTRUCTION

BINTRAP applies its perturbations directly to stripped decompiled pseudocode (§3.4). This section examines whether the resulting inputs are ones that an analyst could actually encounter. Section B.1 tests which perturbations survive a real source-to-pseudocode pipeline, Section B.2 measures how often binaries in the wild retain the symbols that the renaming vectors rely on, and Section B.3 discusses why we construct the benchmark at the pseudocode level.

B.1 SURVIVAL AND EQUIVALENCE THROUGH COMPILATION, STRIPPING, AND DECOMPILATION

Protocol. We sample 50 functions from BINTRAP and apply each of the four perturbations to their C/C++ source rather than to their pseudocode. Each source-level perturbation is first injected automatically and then manually corrected for any inconsistencies it introduces. We compile each perturbed source with GCC at O2, produce both a stripped build and an unstripped build that retains the symbol table and debug information, decompile both with Ghidra, and compare the resulting pseudocode with the pseudocode-level variant that BINTRAP constructs for the same function. We measure three properties:

- **Appears:** the number of functions whose pseudocode still contains the injected tokens.
- **Same role:** the number of injected tokens that appear in the same syntactic position as in the pseudocode-level variant, namely a call argument for P_{str}, a callee or function name for P_{func}, a declaration or parameter for P_{var}, and a type for P_{type}.
- **Surrounding code unchanged:** the number of functions whose perturbed stripped build decompiles to the same pseudocode as the unperturbed stripped build, apart from the injected tokens themselves.

Table 3: Survival of source-level perturbations through compilation (GCC, O2), stripping, and re-decompilation (Ghidra) on 50 sampled functions. Unstripped builds retain the symbol table and debug information. Appears counts functions out of 50; Same role counts injected tokens.

		Stripped		**Unstripped**		**Surrounding**
Vector	**Source-level change**	Appears	Same role	Appears	Same role	**code unchanged**
P_{str}	insert topic print statements	50/50	100/100	50/50	100/100	50/50
P_{func}	rename function and callees	0/50	0/144	50/50	144/144	50/50
P_{var}	rename locals and parameters	0/50	0/198	50/50	165/198	50/50
P_{type}	rewrite declared types	0/50	0/127	50/50	104/127	47/50

Results. Table 3 shows that P_{str} survives the full chain. Every injected string reaches the pseudocode of all 50 functions, in the same call-argument position that the pseudocode-level variant uses, and the rest of the function decompiles to the same statements as the unperturbed binary in all 50 cases. Source-level string injection and our pseudocode-level construction therefore yield equivalent inputs. The injected statements are output calls only: they read and write no program state, so no branch, return value, or data transformation changes, and the only difference in behavior is the diagnostic text they print, which is exactly the artifact a developer adding logging would produce.

P_{func}, P_{var}, and P_{type} do not survive stripping, since stripping removes the symbol table and debug information that carry them. In the unstripped build, however, they all reach the pseudocode: every renamed function name appears in its original role, and most renamed variables and types do as well (165/198 and 104/127). The unmatched names are present in the debug information, and O2 leaves no corresponding variable or type in the binary for the decompiler to attach them to. In addition, although optimization leaves some types unrecoverable in three P_{type} functions, so that the affected statements are decompiled without their struct form, the injected type names still reach the pseudocode in their original role. For the identifiers that reach the pseudocode, source-level renaming and our pseudocode-level construction yield equivalent inputs. These three vectors therefore apply to binaries that retain symbols.

In summary, the above results show that (1) the perturbations survive a real source-to-pseudocode pipeline, and (2) applying them directly to pseudocode is nearly equivalent to applying them at the source level, so that the inputs BINTRAP evaluates are ones an analyst could encounter on a real binary.

B.2 PREVALENCE OF SYMBOL-BEARING BINARIES

The renaming vectors are relevant only if analysts encounter binaries that retain symbols. To estimate how common this is, we collect the ELF samples submitted to MalwareBazaar between August 2025 and August 2026, deduplicate them by their VirusTotal `vhash`, and check whether each distinct sample retains its symbol table. Of the 45,403 distinct samples, 25,445 (56.0%) are unstripped. Symbol-bearing binaries are therefore the majority case in this population, and analyzing them is one of the main use cases of LLM-based binary understanding, where function names, variable names, and types are directly visible to the recovery system.

Keeping symbols can also be a deliberate choice. If misleading strings already steer an LLM toward a wrong interpretation, a developer who wants that outcome has an incentive to keep the symbol table and fill it with function names, variable names, and types that support the same false interpretation, rather than removing them. The renaming vectors measure how much additional influence such a developer gains.

B.3 CONSTRUCTING PERTURBATIONS AT THE PSEUDOCODE LEVEL

We construct the perturbed variants at the pseudocode level because the end-to-end alternative does not scale. That alternative injects each perturbation into the source code and rebuilds the binary, for example by extending a binary dataset construction pipeline such as Assemblage (Liu et al., 2024). It requires every perturbed source to compile and a separate build for every function, topic, and perturbation vector, which amounts to 25,000 builds for BINTRAP, and it exposes the

construction to failures that are unrelated to robustness, such as build errors introduced by renamed declarations. AST rewriting on the stripped pseudocode avoids both, is deterministic, and runs over the full benchmark. Section B.1 validates the substitution on a sample of 50 functions, where the end-to-end construction yields nearly equivalent inputs. The sample therefore supports extending the pseudocode-level construction to the full benchmark. In addition, extending end-to-end construction to the full benchmark remains a promising direction for future work.

C Reference-Based Metrics

For each semantic recovery task, we follow the reference-based evaluation used in prior work. This appendix states these metrics in full. None of them can express whether a prediction has been pulled toward a misleading topic, which is what the two-axis judge protocol of Section 3.5 measures. Throughout, y denotes the reference answer of an evaluation instance and $\hat{y}$ the corresponding prediction.

C.1 T1: Token F1

Following SymGen (Jiang et al., 2025) and SymLM (Jin et al., 2022), a predicted function name is compared with the reference at the granularity of name words rather than as a whole string. Both names are split into their constituent words, which yields a reference token set $W = \{w_1, \ldots, w_n\}$ and a predicted token set $\hat{W} = \{\hat{w}_1, \ldots, \hat{w}_m\}$. Let $\hat{w} \simeq w$ denote that two tokens carry the same meaning, and define the matching indicator

$$\mathbf{1}_W(\hat{w}) = \begin{cases} 1 & \text{if } \exists\, w \in W \text{ such that } \hat{w} \simeq w, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

The token-level true positives, false positives, and false negatives of a single function are

$$\mathrm{TP} = \sum_{\hat{w} \in \hat{W}} \mathbf{1}_W(\hat{w}), \qquad \mathrm{FP} = \sum_{\hat{w} \in \hat{W}} \big(1 - \mathbf{1}_W(\hat{w})\big), \qquad \mathrm{FN} = |W| - \mathrm{TP}, \tag{2}$$

from which

$$\mathrm{P} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \qquad \mathrm{R} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \qquad \mathrm{F1} = \frac{2 \cdot \mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}}. \tag{3}$$

We report the average of these per-function scores over the benchmark. The relation $\simeq$ is the semantic equivalence induced by the CodeWordNet word clusters of SymLM (Jin et al., 2022), which SymGen (Jiang et al., 2025) also adopts: word embeddings trained on a code corpus place synonyms and morphological variants in the same cluster, and tokens within a cluster are treated as matching. Taking $\simeq$ to be string equality recovers strict token overlap. As an illustration, the reference `ftp_timeleft_accept` has words $W = \{\textit{ftp}, \textit{time}, \textit{left}, \textit{accept}\}$, and a prediction with words $\hat{W} = \{\textit{get}, \textit{time}, \textit{left}\}$ gives $\mathrm{TP} = 2$, $\mathrm{FP} = 1$, and $\mathrm{FN} = 2$, yielding $\mathrm{P} = 0.67$, $\mathrm{R} = 0.50$, and $\mathrm{F1} = 0.57$.

C.2 T2: Exact Match and Token F1

Let V denote the set of decompiler-generated variables in the benchmark, and let n_v and $\hat{n}_v$ be the reference and predicted name of a variable $v \in V$. Following the perfect-match criterion of ReSym (Xie et al., 2024), which coincides with the top-1 accuracy of VarBERT (Pal et al., 2024), exact match is the fraction of variables whose predicted name reproduces the reference name exactly,

$$\mathrm{EM} = \frac{1}{|V|} \sum_{v \in V} \mathbf{1}\big[\hat{n}_v = n_v\big], \qquad |V| = \sum_{d} |V_d|, \tag{4}$$

where V_d are the variables of decompiled function d. This criterion is deliberately strict: a prediction that is semantically equivalent to the reference but spelled differently, such as `size` for `bufferSize`, counts as incorrect (Xie et al., 2024). We therefore also report the token-level metric of GenNm (Xu et al., 2025), which tokenizes each name pair into W_v and $\hat{W}_v$ as in T1 and compares the two sets up to the same semantic equivalence,

$$\mathrm{P}_v = \frac{\big|\{\hat{w} \in \hat{W}_v : \exists\, w \in W_v,\ \hat{w} \simeq w\}\big|}{|\hat{W}_v|}, \qquad \mathrm{R}_v = \frac{\big|\{w \in W_v : \exists\, \hat{w} \in \hat{W}_v,\ \hat{w} \simeq w\}\big|}{|W_v|}, \tag{5}$$

with $\mathrm{F1}_v$ the harmonic mean of P_v and R_v; we report the mean over $v \in V$. GenNm reuses the word clusters of SymLM together with edit-distance rules to decide $\simeq$ (Xu et al., 2025).

C.3 T3: BLEU

Following MiSum (Zhu et al., 2025), generated summaries are scored with BLEU (Papineni et al., 2002) against the reference summary. For n-gram order n, let $G_n(\hat{y})$ be the n-grams of a generated summary and let $c_{\hat{y}}(g)$ and $c_y(g)$ be the number of occurrences of the n-gram g in the generated and reference summary. The modified n-gram precision over the benchmark $\mathcal{B} = \{(\hat{y}, y)\}$ clips every count by the count available in the reference,

$$p_n = \frac{\sum_{(\hat{y},y)\in\mathcal{B}} \sum_{g\in G_n(\hat{y})} \min\big(c_{\hat{y}}(g),\, c_y(g)\big)}{\sum_{(\hat{y},y)\in\mathcal{B}} \sum_{g\in G_n(\hat{y})} c_{\hat{y}}(g)}. \tag{6}$$

Writing $c = \sum_{(\hat{y},y)\in\mathcal{B}} |\hat{y}|$ and $r = \sum_{(\hat{y},y)\in\mathcal{B}} |y|$ for the total candidate and reference lengths, the brevity penalty and the final score are

$$\mathrm{BP} = \begin{cases} 1 & \text{if } c > r, \\ e^{1-r/c} & \text{if } c \le r, \end{cases} \qquad \mathrm{BLEU} = \mathrm{BP} \cdot \exp\Big(\sum_{n=1}^{N} w_n \log p_n\Big), \tag{7}$$

where $N = 4$ and $w_n = 1/4$, so that the score is the geometric mean of the unigram through 4-gram precisions, penalized for summaries shorter than their reference.

C.4 T4: Edit Similarity

Following LLM4Decompile (Tan et al., 2024), the reconstructed source code is compared with the reference source by edit similarity. Let $\mathrm{lev}(\hat{y}, y)$ be the Levenshtein distance between the two character sequences, that is, the minimum number of character insertions, deletions, and substitutions that transform one into the other. We compute edit similarity as this distance normalized by the length of the longer sequence, which places it in $[0, 1]$,

$$\mathrm{ES}(\hat{y}, y) = 1 - \frac{\mathrm{lev}(\hat{y}, y)}{\max\big(|\hat{y}|,\, |y|\big)}, \tag{8}$$

so that $\mathrm{ES} = 1$ exactly when the reconstruction is character-identical to the reference. We report the mean of ES over the benchmark.

D Full Evaluation Results

This section reports the complete numerical results behind the main-text figures and tables. Unless stated otherwise, all values are averaged over the ten misleading topics. In Appendices D–F, shading marks mislead rates of at least 10% (yellow), 30% (orange), and 50% (red).

D.1 Results Judged by Claude Sonnet 4.6

Tables 4–7 report accuracy and mislead rate for every system and perturbation setting, scored with Claude Sonnet 4.6 as the judge. These are the results visualized in Figure 2. Claude Sonnet 4.6 also appears in each table as one of the evaluated systems. The judge scores its predictions with the same rubric as those of every other system.

Under $\mathrm{P}_{\mathrm{comb}}$, mislead rate on T1 ranges from 11.2% for Claude Opus 4.6 to 77.0% for SymGen. On T2, mislead rate stays below 30% for every system and setting. On T3, $\mathrm{P}_{\mathrm{func}}$ and $\mathrm{P}_{\mathrm{comb}}$ give the same average mislead rate (44.5%). On T4, the average mislead rate exceeds 50% under $\mathrm{P}_{\mathrm{func}}$, $\mathrm{P}_{\mathrm{str}}$, and $\mathrm{P}_{\mathrm{comb}}$.

Table 4: T1 function-name recovery: accuracy and mislead rate (%) with Claude Sonnet 4.6 as the judge, averaged over the 10 topics.

Category	System	Orig. Acc / Mis	P_{var} Acc / Mis	P_{func} Acc / Mis	P_{type} Acc / Mis	P_{str} Acc / Mis	P_{comb} Acc / Mis
Specialized	DeGPT	9.5 / 0.4	5.8 / 36.1	5.4 / 32.3	6.6 / 41.7	4.8 / 63.8	2.3 / 59.7
	SymGen	4.5 / 0.1	3.0 / 20.6	0.9 / 78.8	2.1 / 60.2	1.7 / 19.1	0.7 / 77.0
General LLM	Claude Opus 4.6	18.9 / 0.7	14.3 / 8.5	14.7 / 20.3	15.7 / 7.0	12.3 / 4.7	10.9 / 11.2
	Claude Sonnet 4.6	14.2 / 0.6	10.9 / 12.6	10.4 / 15.8	11.4 / 12.6	10.3 / 14.2	7.7 / 14.4
	Claude Haiku 4.5	9.5 / 0.6	6.9 / 23.6	6.5 / 22.6	7.4 / 26.1	5.4 / 33.7	3.6 / 39.5
	DeepSeek v3.2	9.3 / 0.4	8.0 / 19.8	5.8 / 28.4	7.8 / 22.0	6.2 / 31.1	4.3 / 37.6
	GLM 5	10.0 / 0.4	8.8 / 12.0	8.0 / 18.3	9.5 / 16.8	8.3 / 12.4	6.8 / 14.7
	Kimi K2.5	10.9 / 0.7	8.8 / 17.8	7.7 / 21.5	9.1 / 17.8	8.4 / 19.0	6.2 / 20.4
	Qwen3-Coder-Next	6.6 / 0.3	4.5 / 41.7	3.7 / 36.1	4.6 / 39.0	1.4 / 89.6	1.4 / 75.9

Table 5: T2 variable-name recovery: accuracy and mislead rate (%) with Claude Sonnet 4.6 as the judge, averaged over the 10 topics.

Category	System	Orig. Acc / Mis	P_{var} Acc / Mis	P_{func} Acc / Mis	P_{type} Acc / Mis	P_{str} Acc / Mis	P_{comb} Acc / Mis
Specialized	GenNm	1.4 / 0.0	0.2 / 2.1	1.3 / 3.0	0.8 / 2.6	0.4 / 2.2	0.7 / 3.6
	ReSym	11.2 / 0.1	6.2 / 28.1	8.9 / 12.0	11.0 / 11.4	6.0 / 25.5	5.7 / 25.2
	VarBERT	14.3 / 0.8	13.5 / 2.1	7.8 / 3.4	6.6 / 3.8	7.8 / 6.6	6.1 / 4.3
General LLM	Claude Opus 4.6	38.5 / 0.1	40.6 / 7.1	36.1 / 2.7	38.6 / 0.7	36.6 / 0.8	35.8 / 3.6
	Claude Sonnet 4.6	34.8 / 0.1	34.3 / 8.5	31.6 / 3.0	34.2 / 1.3	32.4 / 1.0	29.5 / 3.9
	Claude Haiku 4.5	26.4 / 0.0	23.2 / 9.4	23.3 / 4.5	24.6 / 4.2	22.1 / 3.4	17.5 / 7.5
	DeepSeek v3.2	24.1 / 0.1	22.8 / 9.4	20.0 / 7.9	21.7 / 6.5	19.9 / 5.8	16.9 / 10.0
	GLM 5	27.8 / 0.0	26.2 / 8.3	24.9 / 3.2	26.9 / 1.5	24.4 / 1.1	21.5 / 4.7
	Kimi K2.5	25.8 / 0.2	21.9 / 11.8	22.6 / 5.1	23.5 / 3.1	24.5 / 1.4	17.7 / 5.5
	Qwen3-Coder-Next	21.7 / 0.1	14.2 / 17.1	17.5 / 8.9	17.6 / 11.3	12.6 / 20.8	8.8 / 21.4

Table 6: T3 function summarization: accuracy and mislead rate (%) with Claude Sonnet 4.6 as the judge, averaged over the 10 topics.

Category	System	Orig. Acc / Mis	P_{var} Acc / Mis	P_{func} Acc / Mis	P_{type} Acc / Mis	P_{str} Acc / Mis	P_{comb} Acc / Mis
General LLM	Claude Opus 4.6	41.0 / 0.1	33.6 / 12.6	27.9 / 36.0	38.9 / 4.2	35.6 / 6.1	24.7 / 22.9
	Claude Sonnet 4.6	34.5 / 0.1	27.9 / 14.3	22.9 / 40.2	30.7 / 9.3	27.8 / 11.1	16.5 / 28.2
	Claude Haiku 4.5	22.1 / 0.3	13.4 / 29.4	11.5 / 48.7	15.1 / 27.2	8.3 / 36.5	4.5 / 57.2
	DeepSeek v3.2	20.6 / 0.3	15.8 / 21.5	11.2 / 42.8	16.3 / 18.6	10.8 / 33.4	7.0 / 46.9
	GLM 5	24.1 / 0.1	18.7 / 14.0	14.6 / 38.8	20.2 / 11.3	16.4 / 16.0	10.7 / 29.7
	Kimi K2.5	26.3 / 0.8	18.9 / 27.2	15.2 / 48.2	21.8 / 22.4	18.2 / 17.8	11.0 / 38.7
	Qwen3-Coder-Next	17.0 / 0.2	8.2 / 49.8	6.7 / 57.0	10.5 / 41.4	0.9 / 91.4	1.6 / 87.8

Table 7: T4 decompilation-to-source: accuracy and mislead rate (%) with Claude Sonnet 4.6 as the judge, averaged over the 10 topics.

Category	System	Orig. Acc / Mis	P_{var} Acc / Mis	P_{func} Acc / Mis	P_{type} Acc / Mis	P_{str} Acc / Mis	P_{comb} Acc / Mis
Specialized	LLM4Decompile	0.3 / 0.6	0.4 / 44.9	0.0 / 82.3	0.2 / 77.1	0.1 / 79.8	0.0 / 90.6
	SK2Decompile	0.3 / 0.6	0.1 / 2.0	0.0 / 72.6	0.0 / 18.9	0.0 / 76.8	0.0 / 78.5

continued on next page

Table 7: (continued)

Category	System	Orig. Acc / Mis	P_{var} Acc / Mis	P_{func} Acc / Mis	P_{type} Acc / Mis	P_{str} Acc / Mis	P_{comb} Acc / Mis
General LLM	Claude Opus 4.6	9.3 / 0.8	6.9 / 8.4	5.3 / 47.6	6.5 / 6.8	4.3 / 26.2	2.9 / 29.1
	Claude Sonnet 4.6	7.8 / 0.7	6.0 / 15.9	3.3 / 52.1	6.3 / 15.8	3.7 / 28.1	1.9 / 33.3
	Claude Haiku 4.5	2.5 / 0.7	1.4 / 22.3	0.8 / 56.2	1.4 / 28.8	0.2 / 69.0	0.2 / 60.8
	DeepSeek v3.2	2.1 / 0.3	1.6 / 17.4	0.8 / 47.4	1.2 / 23.7	0.6 / 47.2	0.5 / 42.2
	GLM 5	3.0 / 0.4	2.2 / 11.4	1.3 / 42.8	2.3 / 17.1	0.8 / 34.7	0.5 / 30.5
	Kimi K2.5	3.1 / 0.8	2.4 / 20.6	1.3 / 54.5	2.1 / 28.8	1.2 / 41.5	1.1 / 45.0
	Qwen3-Coder-Next	1.0 / 0.4	0.5 / 54.2	0.2 / 71.7	0.5 / 65.9	0.0 / 89.9	0.0 / 81.9

D.2 Confidence Intervals

Each rate in Appendix D.1 aggregates the predictions of 500 functions under 10 misleading topics. We estimate its sampling error by resampling the 500 functions with replacement over 2,000 bootstrap replicates, carrying all 10 topics of a resampled function together, and report the 95% interval as a half-width around the observed rate. Tables 8 and 9 give the intervals for accuracy and for mislead rate.

On T1, T3, and T4, the mislead rate under P_{func} and P_{comb} rises by 10.5 to 90.0 points over the original inputs, against half-widths of 0.8 to 3.1 points. Under P_{comb}, the accuracy interval does not overlap the original one for every system except GLM 5 on T1 and LLM4Decompile and SK2Decompile on T4, whose original accuracy is only 0.3%. On T2, several changes fall inside the intervals, which matches the finding that variable-name recovery is the least affected task.

Table 8: Accuracy (%) with bootstrap 95% confidence intervals, averaged over the 10 topics.

Task	System	Orig.	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T1	SymGen	$4.5_{\pm1.8}$	$3.0_{\pm1.4}$	$0.9_{\pm0.8}$	$2.1_{\pm1.2}$	$1.7_{\pm1.1}$	$0.7_{\pm0.6}$
	DeGPT	$9.5_{\pm2.3}$	$5.8_{\pm1.8}$	$5.4_{\pm1.7}$	$6.6_{\pm1.8}$	$4.8_{\pm1.6}$	$2.3_{\pm1.0}$
	Claude Opus 4.6	$18.9_{\pm3.4}$	$14.3_{\pm2.7}$	$14.7_{\pm2.7}$	$15.7_{\pm2.9}$	$12.3_{\pm2.5}$	$10.9_{\pm2.4}$
	Claude Sonnet 4.6	$14.2_{\pm2.9}$	$10.9_{\pm2.4}$	$10.4_{\pm2.5}$	$11.4_{\pm2.5}$	$10.3_{\pm2.3}$	$7.7_{\pm2.0}$
	Claude Haiku 4.5	$9.5_{\pm2.2}$	$6.9_{\pm1.9}$	$6.5_{\pm1.9}$	$7.4_{\pm2.0}$	$5.4_{\pm1.8}$	$3.6_{\pm1.4}$
	GLM 5	$10.0_{\pm2.2}$	$8.8_{\pm2.1}$	$8.0_{\pm2.0}$	$9.5_{\pm2.0}$	$8.3_{\pm1.9}$	$6.8_{\pm1.7}$
	Kimi K2.5	$10.9_{\pm2.2}$	$8.8_{\pm2.0}$	$7.7_{\pm2.0}$	$9.1_{\pm2.1}$	$8.4_{\pm2.0}$	$6.2_{\pm1.7}$
	DeepSeek v3.2	$9.3_{\pm2.0}$	$8.0_{\pm1.9}$	$5.8_{\pm1.7}$	$7.8_{\pm1.9}$	$6.2_{\pm1.7}$	$4.3_{\pm1.4}$
	Qwen3-Coder-Next	$6.6_{\pm2.0}$	$4.5_{\pm1.7}$	$3.7_{\pm1.5}$	$4.6_{\pm1.7}$	$1.4_{\pm1.1}$	$1.4_{\pm0.9}$
T2	GenNm	$1.4_{\pm0.5}$	$0.2_{\pm0.2}$	$1.3_{\pm0.4}$	$0.8_{\pm0.2}$	$0.4_{\pm0.2}$	$0.7_{\pm0.4}$
	ReSym	$11.2_{\pm1.5}$	$6.2_{\pm3.1}$	$8.9_{\pm1.1}$	$11.0_{\pm1.2}$	$6.0_{\pm0.9}$	$5.7_{\pm1.2}$
	VarBERT	$14.3_{\pm2.4}$	$13.5_{\pm1.9}$	$7.8_{\pm1.5}$	$6.6_{\pm1.2}$	$7.8_{\pm1.4}$	$6.1_{\pm1.1}$
	Claude Opus 4.6	$38.5_{\pm6.3}$	$40.6_{\pm5.6}$	$36.1_{\pm5.9}$	$38.6_{\pm6.2}$	$36.6_{\pm5.8}$	$35.8_{\pm5.2}$
	Claude Sonnet 4.6	$34.8_{\pm5.6}$	$34.3_{\pm4.9}$	$31.6_{\pm5.2}$	$34.2_{\pm5.5}$	$32.4_{\pm5.4}$	$29.5_{\pm4.4}$
	Claude Haiku 4.5	$26.4_{\pm4.5}$	$23.2_{\pm3.7}$	$23.3_{\pm4.1}$	$24.6_{\pm4.2}$	$22.1_{\pm3.7}$	$17.5_{\pm2.8}$
	GLM 5	$27.8_{\pm4.2}$	$26.2_{\pm3.8}$	$24.9_{\pm4.0}$	$26.9_{\pm3.9}$	$24.4_{\pm3.8}$	$21.5_{\pm3.1}$
	Kimi K2.5	$25.8_{\pm4.4}$	$21.9_{\pm3.1}$	$22.6_{\pm3.9}$	$23.5_{\pm4.0}$	$24.5_{\pm4.1}$	$17.7_{\pm2.8}$
	DeepSeek v3.2	$24.1_{\pm4.4}$	$22.8_{\pm3.8}$	$20.0_{\pm3.8}$	$21.7_{\pm4.0}$	$19.9_{\pm3.8}$	$16.9_{\pm3.1}$
	Qwen3-Coder-Next	$21.7_{\pm4.1}$	$14.2_{\pm3.4}$	$17.5_{\pm3.4}$	$17.6_{\pm3.6}$	$12.6_{\pm3.1}$	$8.8_{\pm2.4}$
T3	Claude Opus 4.6	$41.0_{\pm4.0}$	$33.6_{\pm3.8}$	$27.9_{\pm3.4}$	$38.9_{\pm3.7}$	$35.6_{\pm3.7}$	$24.7_{\pm3.1}$
	Claude Sonnet 4.6	$34.5_{\pm3.8}$	$27.9_{\pm3.5}$	$22.9_{\pm3.2}$	$30.7_{\pm3.7}$	$27.8_{\pm3.5}$	$16.5_{\pm2.8}$
	Claude Haiku 4.5	$22.1_{\pm3.3}$	$13.4_{\pm2.6}$	$11.5_{\pm2.4}$	$15.1_{\pm2.8}$	$8.3_{\pm2.0}$	$4.5_{\pm1.5}$
	GLM 5	$24.1_{\pm3.3}$	$18.7_{\pm2.9}$	$14.6_{\pm2.5}$	$20.2_{\pm3.0}$	$16.4_{\pm2.6}$	$10.7_{\pm2.2}$
	Kimi K2.5	$26.3_{\pm3.5}$	$18.9_{\pm3.0}$	$15.2_{\pm2.5}$	$21.8_{\pm3.0}$	$18.2_{\pm2.8}$	$11.0_{\pm2.3}$
	DeepSeek v3.2	$20.6_{\pm3.3}$	$15.8_{\pm2.8}$	$11.2_{\pm2.4}$	$16.3_{\pm3.0}$	$10.8_{\pm2.4}$	$7.0_{\pm1.9}$
	Qwen3-Coder-Next	$17.0_{\pm2.9}$	$8.2_{\pm2.3}$	$6.7_{\pm1.9}$	$10.5_{\pm2.4}$	$0.9_{\pm0.9}$	$1.6_{\pm0.9}$

continued on next page

Table 8: (continued)

Task	System	Orig.	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T4	LLM4Decompile	$0.3_{\pm 0.5}$	$0.4_{\pm 0.4}$	$0.0_{\pm 0.0}$	$0.2_{\pm 0.3}$	$0.1_{\pm 0.1}$	$0.0_{\pm 0.0}$
	SK2Decompile	$0.3_{\pm 0.5}$	$0.1_{\pm 0.3}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.1}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
	Claude Opus 4.6	$9.3_{\pm 2.4}$	$6.9_{\pm 1.9}$	$5.3_{\pm 1.6}$	$6.5_{\pm 1.8}$	$4.3_{\pm 1.4}$	$2.9_{\pm 1.1}$
	Claude Sonnet 4.6	$7.8_{\pm 2.0}$	$6.0_{\pm 1.7}$	$3.3_{\pm 1.1}$	$6.3_{\pm 1.6}$	$3.7_{\pm 1.2}$	$1.9_{\pm 0.9}$
	Claude Haiku 4.5	$2.5_{\pm 1.2}$	$1.4_{\pm 0.9}$	$0.8_{\pm 0.7}$	$1.4_{\pm 0.9}$	$0.2_{\pm 0.5}$	$0.2_{\pm 0.2}$
	GLM 5	$3.0_{\pm 1.1}$	$2.2_{\pm 1.0}$	$1.3_{\pm 0.8}$	$2.3_{\pm 1.0}$	$0.8_{\pm 0.5}$	$0.5_{\pm 0.4}$
	Kimi K2.5	$3.1_{\pm 1.3}$	$2.4_{\pm 0.9}$	$1.3_{\pm 0.7}$	$2.1_{\pm 0.9}$	$1.2_{\pm 0.7}$	$1.1_{\pm 0.5}$
	DeepSeek v3.2	$2.1_{\pm 1.0}$	$1.6_{\pm 0.7}$	$0.8_{\pm 0.6}$	$1.2_{\pm 0.7}$	$0.6_{\pm 0.5}$	$0.5_{\pm 0.5}$
	Qwen3-Coder-Next	$1.0_{\pm 0.8}$	$0.5_{\pm 0.6}$	$0.2_{\pm 0.6}$	$0.5_{\pm 0.5}$	$0.0_{\pm 0.1}$	$0.0_{\pm 0.0}$

Table 9: Mislead rate (%) with bootstrap 95% confidence intervals, averaged over the 10 topics.

Task	System	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T1	SymGen	$20.6_{\pm 2.4}$	$78.8_{\pm 1.9}$	$60.2_{\pm 2.6}$	$19.1_{\pm 2.3}$	$77.0_{\pm 2.0}$
	DeGPT	$36.1_{\pm 2.4}$	$32.3_{\pm 2.5}$	$41.7_{\pm 2.7}$	$63.8_{\pm 3.1}$	$59.7_{\pm 2.5}$
	Claude Opus 4.6	$8.5_{\pm 1.4}$	$20.3_{\pm 2.1}$	$7.0_{\pm 1.1}$	$4.7_{\pm 1.2}$	$11.2_{\pm 2.0}$
	Claude Sonnet 4.6	$12.6_{\pm 1.7}$	$15.8_{\pm 1.7}$	$12.6_{\pm 1.6}$	$14.2_{\pm 2.2}$	$14.4_{\pm 2.1}$
	Claude Haiku 4.5	$23.6_{\pm 2.0}$	$22.6_{\pm 2.0}$	$26.1_{\pm 2.1}$	$33.7_{\pm 3.0}$	$39.5_{\pm 2.6}$
	GLM 5	$12.0_{\pm 1.4}$	$18.3_{\pm 1.7}$	$16.8_{\pm 1.6}$	$12.4_{\pm 1.8}$	$14.7_{\pm 1.7}$
	Kimi K2.5	$17.8_{\pm 1.8}$	$21.5_{\pm 1.8}$	$17.8_{\pm 1.8}$	$19.0_{\pm 2.3}$	$20.4_{\pm 2.2}$
	DeepSeek v3.2	$19.8_{\pm 1.9}$	$28.4_{\pm 2.0}$	$22.0_{\pm 2.1}$	$31.1_{\pm 2.4}$	$37.6_{\pm 2.2}$
	Qwen3-Coder-Next	$41.7_{\pm 2.5}$	$36.1_{\pm 2.3}$	$39.0_{\pm 2.3}$	$89.6_{\pm 2.0}$	$75.9_{\pm 2.3}$
T2	GenNm	$2.1_{\pm 1.3}$	$3.0_{\pm 0.5}$	$2.6_{\pm 0.5}$	$2.2_{\pm 0.4}$	$3.6_{\pm 1.1}$
	ReSym	$28.1_{\pm 6.5}$	$12.0_{\pm 1.0}$	$11.4_{\pm 0.9}$	$25.5_{\pm 1.1}$	$25.2_{\pm 2.2}$
	VarBERT	$2.1_{\pm 0.4}$	$3.4_{\pm 0.4}$	$3.8_{\pm 0.4}$	$6.6_{\pm 0.6}$	$4.3_{\pm 0.5}$
	Claude Opus 4.6	$7.1_{\pm 1.3}$	$2.7_{\pm 0.8}$	$0.7_{\pm 0.3}$	$0.8_{\pm 1.1}$	$3.6_{\pm 0.8}$
	Claude Sonnet 4.6	$8.5_{\pm 1.6}$	$3.0_{\pm 0.8}$	$1.3_{\pm 0.3}$	$1.0_{\pm 1.1}$	$3.9_{\pm 1.1}$
	Claude Haiku 4.5	$9.4_{\pm 1.8}$	$4.5_{\pm 1.1}$	$4.2_{\pm 1.0}$	$3.4_{\pm 1.3}$	$7.5_{\pm 1.3}$
	GLM 5	$8.3_{\pm 1.7}$	$3.2_{\pm 0.8}$	$1.5_{\pm 0.4}$	$1.1_{\pm 0.8}$	$4.7_{\pm 1.0}$
	Kimi K2.5	$11.8_{\pm 1.8}$	$5.1_{\pm 1.1}$	$3.1_{\pm 0.7}$	$1.4_{\pm 0.8}$	$5.5_{\pm 1.1}$
	DeepSeek v3.2	$9.4_{\pm 1.8}$	$7.9_{\pm 1.8}$	$6.5_{\pm 1.4}$	$5.8_{\pm 1.7}$	$10.0_{\pm 1.8}$
	Qwen3-Coder-Next	$17.1_{\pm 2.0}$	$8.9_{\pm 1.4}$	$11.3_{\pm 1.5}$	$20.8_{\pm 2.6}$	$21.4_{\pm 2.4}$
T3	Claude Opus 4.6	$12.6_{\pm 1.8}$	$36.0_{\pm 3.0}$	$4.2_{\pm 0.8}$	$6.1_{\pm 1.9}$	$22.9_{\pm 2.6}$
	Claude Sonnet 4.6	$14.3_{\pm 1.8}$	$40.2_{\pm 3.1}$	$9.3_{\pm 1.2}$	$11.1_{\pm 2.3}$	$28.2_{\pm 2.7}$
	Claude Haiku 4.5	$29.4_{\pm 2.3}$	$48.7_{\pm 3.0}$	$27.2_{\pm 1.9}$	$36.5_{\pm 2.8}$	$57.2_{\pm 2.6}$
	GLM 5	$14.0_{\pm 1.5}$	$38.8_{\pm 2.9}$	$11.3_{\pm 1.3}$	$16.0_{\pm 2.3}$	$29.7_{\pm 2.7}$
	Kimi K2.5	$27.2_{\pm 2.0}$	$48.2_{\pm 2.9}$	$22.4_{\pm 1.8}$	$17.8_{\pm 2.3}$	$38.7_{\pm 2.6}$
	DeepSeek v3.2	$21.5_{\pm 1.8}$	$42.8_{\pm 3.1}$	$18.6_{\pm 1.6}$	$33.4_{\pm 2.5}$	$46.9_{\pm 2.8}$
	Qwen3-Coder-Next	$49.8_{\pm 2.5}$	$57.0_{\pm 3.0}$	$41.4_{\pm 2.2}$	$91.4_{\pm 1.8}$	$87.8_{\pm 2.1}$
T4	LLM4Decompile	$44.9_{\pm 2.3}$	$82.3_{\pm 1.4}$	$77.1_{\pm 1.7}$	$79.8_{\pm 1.7}$	$90.6_{\pm 0.8}$
	SK2Decompile	$2.0_{\pm 1.0}$	$72.6_{\pm 1.9}$	$18.9_{\pm 1.4}$	$76.8_{\pm 1.8}$	$78.5_{\pm 2.2}$
	Claude Opus 4.6	$8.4_{\pm 1.5}$	$47.6_{\pm 2.8}$	$6.8_{\pm 1.2}$	$26.2_{\pm 3.4}$	$29.1_{\pm 3.0}$
	Claude Sonnet 4.6	$15.9_{\pm 1.9}$	$52.1_{\pm 2.9}$	$15.8_{\pm 1.5}$	$28.1_{\pm 3.1}$	$33.3_{\pm 2.8}$
	Claude Haiku 4.5	$22.3_{\pm 1.9}$	$56.2_{\pm 2.4}$	$28.8_{\pm 2.1}$	$69.0_{\pm 2.6}$	$60.8_{\pm 2.8}$
	GLM 5	$11.4_{\pm 1.5}$	$42.8_{\pm 2.1}$	$17.1_{\pm 1.6}$	$34.7_{\pm 2.7}$	$30.5_{\pm 2.3}$
	Kimi K2.5	$20.6_{\pm 2.0}$	$54.5_{\pm 2.5}$	$28.8_{\pm 2.2}$	$41.5_{\pm 3.2}$	$45.0_{\pm 2.6}$
	DeepSeek v3.2	$17.4_{\pm 2.1}$	$47.4_{\pm 2.7}$	$23.7_{\pm 2.2}$	$47.2_{\pm 2.8}$	$42.2_{\pm 2.5}$
	Qwen3-Coder-Next	$54.2_{\pm 2.4}$	$71.7_{\pm 2.3}$	$65.9_{\pm 1.9}$	$89.9_{\pm 2.2}$	$81.9_{\pm 2.4}$

D.3 Reference-Based Results

Table 10 reports the reference-based metrics defined in Appendix C for every system and perturbation setting. Table 2 in the main text averages these values over systems. On T1 and T3, every system

obtains its lowest score under P_{comb}. On T4, edit similarity changes little, and it rises under P_{var} for eight of the nine systems.

Table 10: Reference-based metrics (%) for every system, averaged over the 10 topics. T2 reports exact match / token F1.

Task (metric)	System	Orig.	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T1 (Token F1)	SymGen	10.7	7.6	1.9	5.3	3.8	1.6
	DeGPT	15.6	11.2	10.6	11.2	8.8	5.8
	Claude Opus 4.6	22.2	20.2	19.5	20.8	18.1	17.1
	Claude Sonnet 4.6	20.9	17.3	17.2	18.6	16.5	14.0
	Claude Haiku 4.5	17.0	13.7	13.4	13.8	11.7	8.8
	GLM 5	17.8	15.9	14.7	15.7	14.1	12.4
	Kimi K2.5	16.8	14.2	13.5	14.9	13.7	11.6
	DeepSeek v3.2	15.4	12.8	11.3	13.1	10.4	9.1
	Qwen3-Coder-Next	14.9	11.4	10.4	11.4	5.5	5.3
T2 (EM / Tok. F1)	GenNm	0.2/0.3	0.0/0.0	0.7/2.0	0.2/0.8	0.1/0.7	0.2/0.3
	ReSym	0.6/2.2	0.0/0.0	0.6/2.2	1.1/3.1	2.3/4.5	1.4/2.4
	VarBERT	6.4/6.5	6.2/6.3	2.6/2.7	3.9/3.9	5.2/5.3	2.8/2.9
	Claude Opus 4.6	6.7/8.5	6.0/9.8	3.4/5.4	6.1/8.5	3.2/5.7	4.9/7.8
	Claude Sonnet 4.6	4.3/5.1	4.0/5.4	2.6/3.7	3.6/4.6	2.4/3.7	2.3/3.5
	Claude Haiku 4.5	3.9/6.0	2.0/3.2	1.4/2.7	2.1/3.2	0.6/1.9	0.6/1.2
	GLM 5	3.5/4.8	3.0/4.1	1.5/2.8	2.3/3.8	1.1/2.2	1.1/2.5
	Kimi K2.5	8.7/9.9	9.8/11.2	6.2/7.6	8.5/9.9	7.5/8.5	7.4/8.8
	DeepSeek v3.2	1.8/3.6	1.0/2.7	0.4/1.7	0.8/2.5	0.3/2.2	0.5/1.9
	Qwen3-Coder-Next	2.6/4.1	1.5/2.7	0.6/1.9	1.1/2.3	0.3/1.4	0.4/1.5
T3 (BLEU)	Claude Opus 4.6	14.9	12.8	11.5	13.6	11.0	9.4
	Claude Sonnet 4.6	11.5	9.7	8.7	10.3	7.9	6.5
	Claude Haiku 4.5	6.2	4.7	4.6	5.3	3.6	2.9
	GLM 5	7.3	6.2	5.6	6.6	5.4	4.6
	Kimi K2.5	5.5	4.4	4.1	4.8	4.4	3.4
	DeepSeek v3.2	5.4	4.4	3.9	4.7	3.4	2.8
	Qwen3-Coder-Next	4.2	3.2	2.7	3.2	1.5	1.4
T4 (Edit Sim.)	LLM4Decompile	24.4	24.0	24.9	24.1	21.3	22.7
	SK2Decompile	22.6	23.7	23.4	22.3	21.0	21.3
	Claude Opus 4.6	31.5	33.8	31.1	32.1	32.5	31.3
	Claude Sonnet 4.6	27.9	28.8	25.8	27.0	27.5	26.4
	Claude Haiku 4.5	34.9	35.5	33.3	33.2	26.2	27.4
	GLM 5	29.0	30.4	28.6	29.9	27.1	28.3
	Kimi K2.5	29.2	29.5	28.0	28.4	24.6	24.7
	DeepSeek v3.2	28.6	29.4	28.3	28.6	23.9	24.7
	Qwen3-Coder-Next	28.5	29.0	27.8	28.5	20.5	22.9

D.4 Results Judged by GPT-5.5

We re-score every prediction with GPT-5.5 as the judge, using the same rubric and output schema as the Claude Sonnet 4.6 judge (Appendix A.2) and a temperature of 0. Unlike Claude Sonnet 4.6, GPT-5.5 is not among the evaluated systems. Table 11 reports the full results. As under the Claude Sonnet 4.6 judge, accuracy under P_{comb} is lower than on the original inputs for every system. Appendix E.1 compares the two judges.

Table 11: Accuracy and mislead rate (%) with GPT-5.5 as the judge, averaged over the 10 topics.

Task	System	Orig. Acc	P_{var} Acc	P_{var} Mis	P_{func} Acc	P_{func} Mis	P_{type} Acc	P_{type} Mis	P_{str} Acc	P_{str} Mis	P_{comb} Acc	P_{comb} Mis
	SymGen	5.9	3.4	27.2	1.0	94.0	2.2	71.5	1.9	24.9	0.7	88.3
	DeGPT	13.2	9.2	55.8	8.4	56.8	9.5	60.7	6.3	70.1	3.5	74.3
T1	Claude Opus 4.6	20.5	18.3	36.8	16.5	50.1	19.2	34.5	16.6	24.0	13.8	37.3
	Claude Sonnet 4.6	17.1	13.7	36.4	12.9	43.2	14.8	36.9	12.7	27.9	10.1	36.5
	Claude Haiku 4.5	12.2	9.2	44.1	8.7	47.5	10.0	45.8	7.7	43.2	5.0	56.5
	GLM 5	13.3	10.9	35.4	10.9	43.7	11.7	39.0	9.5	28.5	8.3	37.6
	Kimi K2.5	12.8	10.9	45.8	10.1	50.6	11.7	46.2	10.5	37.7	8.3	47.0
	DeepSeek v3.2	12.3	10.7	46.5	8.7	54.1	10.5	47.1	7.8	46.0	6.0	54.7
	Qwen3-Coder-Next	11.5	8.2	55.6	7.3	58.3	8.2	59.5	3.3	87.8	3.1	83.5
	GenNm	5.2	4.9	9.8	5.1	6.8	4.1	6.0	5.0	4.0	3.6	7.8
	ReSym	14.9	10.8	39.4	11.7	23.1	13.6	23.9	7.7	35.8	7.6	38.0
	VarBERT	17.0	13.2	32.9	11.0	17.6	9.2	20.5	9.7	18.6	7.8	27.4
T2	Claude Opus 4.6	59.1	57.6	68.0	54.6	7.5	58.0	2.4	58.6	1.5	52.7	43.6
	Claude Sonnet 4.6	57.9	51.7	69.2	51.3	8.8	53.8	3.6	53.9	1.5	47.6	48.3
	Claude Haiku 4.5	49.8	39.7	69.1	42.9	10.0	42.9	9.6	41.2	5.0	32.0	51.5
	GLM 5	47.1	40.5	69.5	41.7	8.5	43.1	4.6	41.3	2.6	34.7	46.3
	Kimi K2.5	42.9	31.4	67.8	38.7	11.3	37.1	9.2	40.7	3.1	27.2	45.7
	DeepSeek v3.2	47.6	40.1	70.1	39.3	15.6	40.5	13.5	40.1	9.7	33.0	53.6
	Qwen3-Coder-Next	44.0	35.4	69.1	37.6	14.2	37.1	15.9	31.9	23.0	25.7	59.6
T3	Claude Opus 4.6	38.0	33.9	43.9	28.4	64.1	37.8	32.3	39.3	33.7	29.2	57.4
	Claude Sonnet 4.6	32.7	28.1	50.1	23.4	68.1	30.7	39.8	32.3	40.3	21.1	66.2
	Claude Haiku 4.5	22.3	15.3	59.1	12.9	74.1	17.8	54.7	13.0	63.1	6.7	80.7
	GLM 5	24.2	20.8	47.4	16.2	69.2	21.9	43.2	21.6	51.5	15.0	67.6
	Kimi K2.5	29.5	21.7	54.9	17.0	71.1	24.6	49.8	25.8	47.4	15.9	68.2
	DeepSeek v3.2	22.0	17.8	54.0	13.8	71.6	19.3	48.2	15.2	70.9	10.1	78.2
	Qwen3-Coder-Next	17.5	11.3	63.0	9.2	72.8	12.8	63.2	3.1	94.3	3.2	90.9
	LLM4Decompile	2.8	2.5	73.5	0.1	98.5	1.1	94.5	0.1	94.6	0.1	99.4
	SK2Decompile	0.6	0.9	39.6	0.0	95.7	0.8	70.5	0.0	93.8	0.1	94.5
T4	Claude Opus 4.6	19.2	18.8	34.6	11.0	76.4	20.0	32.4	13.2	44.8	10.7	59.1
	Claude Sonnet 4.6	18.4	16.4	42.8	9.9	75.6	16.9	41.9	12.3	45.7	7.1	64.5
	Claude Haiku 4.5	8.7	7.3	48.8	3.6	81.2	7.5	55.3	1.7	84.5	1.6	82.6
	GLM 5	8.5	9.1	40.1	4.9	72.1	9.4	45.5	4.6	58.5	3.9	62.1
	Kimi K2.5	10.5	9.3	51.2	4.7	81.6	10.0	59.5	6.2	62.8	4.0	76.0
	DeepSeek v3.2	8.1	7.8	52.4	4.2	79.7	7.6	62.9	2.8	79.2	2.3	82.1
	Qwen3-Coder-Next	7.8	6.2	61.5	3.8	79.0	5.9	69.0	0.2	99.1	1.1	90.2

E JUDGE VALIDATION

This section checks the LLM-judge scores against a second judge, against human annotation, and across accuracy thresholds.

E.1 AGREEMENT BETWEEN LLM JUDGES

We compare the scores from the Claude Sonnet 4.6 judge (Appendix D.1) with those from the GPT-5.5 judge (Appendix D.4) cell by cell, where a cell is one evaluated system, task, and perturbation setting. The GPT-5.5 judge is more lenient in absolute terms. It gives a higher accuracy in 201 of the 210 cells and a higher mislead rate in 174 of the 175 perturbed cells. Across these cells, the two judges have a Spearman's ρ of 0.957 for accuracy and 0.868 for mislead rate. The Claude Sonnet 4.6 judge also scores the predictions of Claude Sonnet 4.6 itself. Within each task and setting, Claude Sonnet 4.6 holds the same accuracy rank among the systems under both judges in all 24 cells. Its mislead-rate rank differs in 7 of the 20 perturbed cells, in both directions, so we find no consistent self-preference.

E.2 HUMAN ANNOTATION

Sample. We draw 2,000 predictions, 500 per task, uniformly at random from the full grid of systems $\times$ 5 perturbation settings $\times$ 10 topics, so that both judges and the human annotators are compared on identical inputs.

Protocol. Two of the authors, each with six years of reverse-engineering experience, independently score all 2,000 predictions on both axes using the same 1–5 rubric as the judges (Appendix A.2), blind to the scores of both judges. After independent scoring, the two annotators review every item on which they disagree and resolve it into a single adjudicated label, which serves as the human reference.

Table 12: Agreement of both judges with the adjudicated human labels on the 2,000-prediction sample. Rates are the % of predictions scoring ≥ 4. The Human value is shared by the two judge rows of each task.

			Acc		**Mis**		**Cohen's κ**		**Spearman's ρ**	
Task	n	**Judge**	Human	Judge	Human	Judge	Acc	Mis	Acc	Mis
T1	500	Claude Sonnet 4.6	22.8	10.5	25.1	19.4	0.550	0.819	0.788	0.878
		GPT-5.5		13.4		34.8	0.617	0.661	0.794	0.746
T2	500	Claude Sonnet 4.6	59.2	33.1	5.1	6.1	0.412	0.515	0.625	0.647
		GPT-5.5		52.0		5.9	0.526	0.563	0.656	0.633
T3	500	Claude Sonnet 4.6	23.6	18.6	30.6	23.2	0.659	0.763	0.658	0.932
		GPT-5.5		18.6		47.6	0.623	0.580	0.597	0.846
T4	500	Claude Sonnet 4.6	7.0	1.6	50.0	34.8	0.358	0.663	0.435	0.854
		GPT-5.5		6.1		58.0	0.542	0.722	0.512	0.927

Agreement measures. All scores are binarized at the same threshold as in the main text (≥ 4). Cohen's κ measures item-level agreement between each judge and the adjudicated human labels. Spearman's ρ is computed within each task over the system $\times$ perturbation-setting cells, correlating each judge's cell-level rate of scores ≥ 4 with the annotators' rate for the same cell, so ρ measures agreement on the ordering of systems and settings.

Results. Table 12 reports the results. On the mislead axis, κ ranges from 0.515 to 0.819 for the Claude Sonnet 4.6 judge and from 0.563 to 0.722 for the GPT-5.5 judge, and ρ from 0.647 to 0.932 and from 0.633 to 0.927. On the accuracy axis, κ ranges from 0.358 to 0.659 for the Claude Sonnet 4.6 judge and from 0.526 to 0.623 for the GPT-5.5 judge, and ρ from 0.435 to 0.788 and from 0.512 to 0.794. Compared with the annotators, the Claude Sonnet 4.6 judge gives a lower accuracy rate on every task, and a lower mislead rate on T1, T3, and T4. For the Claude Sonnet 4.6 judge, both κ and ρ are higher on the mislead axis than on the accuracy axis on every task, and the mislead axis is the one our main claims rest on. On T1, T3, and T4, the mislead rates in the main text are therefore conservative relative to the human labels.

E.3 THRESHOLD SENSITIVITY

The main text counts a prediction as accurate when its semantic-accuracy score is at least 4. In the rubric, a score of 3 denotes a generic operational match without domain-specific recovery (Appendix A.2). Table 13 reports the mean accuracy score and the accuracy rate at the more permissive threshold of 3, both with Claude Sonnet 4.6 as the judge. On T1, T3, and T4, both values decrease under every perturbation for every system shown, and the mean score drops by 0.35–0.81 points under P_{comb}. On T2, the changes are smaller and not consistent in sign. The accuracy rates at thresholds 4 and 3 have a Spearman's ρ of 0.91, and the rate at threshold 4 and the mean score have a ρ of 0.87.

Table 13: Mean accuracy score (1–5) / accuracy rate (%) at the ≥ 3 threshold, with Claude Sonnet 4.6 as the judge, averaged over the 10 topics, shown for the specialized tools and three general-purpose LLMs.

Task	System	Orig.	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T1	SymGen	1.40 / 9.1	1.23 / 5.2	1.04 / 0.9	1.18 / 2.9	1.10 / 2.1	1.05 / 0.9
	DeGPT	2.29 / 32.4	1.84 / 20.0	1.81 / 19.1	1.82 / 19.1	1.61 / 12.5	1.48 / 8.7
	Claude Sonnet 4.6	2.31 / 33.8	2.10 / 27.2	2.01 / 25.6	2.15 / 29.1	1.98 / 22.2	1.87 / 19.0
	GLM 5	2.18 / 29.8	2.01 / 24.0	1.91 / 21.5	2.02 / 25.3	1.92 / 20.7	1.82 / 18.1
	Kimi K2.5	2.02 / 25.3	1.82 / 19.1	1.76 / 18.1	1.84 / 20.1	1.79 / 17.2	1.65 / 14.2
T2	GenNm	1.17 / 3.9	1.07 / 1.2	1.20 / 4.4	1.14 / 2.8	1.12 / 2.2	1.14 / 2.5
	ReSym	2.27 / 37.2	1.79 / 18.1	2.13 / 33.1	2.21 / 36.0	1.79 / 20.9	1.80 / 20.8
	VarBERT	2.25 / 29.7	2.30 / 34.0	1.93 / 20.8	1.85 / 16.8	1.87 / 18.2	1.92 / 19.4
	Claude Sonnet 4.6	2.77 / 59.3	2.77 / 62.0	2.67 / 55.6	2.77 / 59.8	2.72 / 58.6	2.70 / 59.3
	GLM 5	2.61 / 56.2	2.58 / 57.1	2.51 / 52.2	2.57 / 54.1	2.51 / 53.0	2.48 / 53.1
	Kimi K2.5	2.47 / 49.2	2.29 / 44.0	2.35 / 44.6	2.38 / 45.3	2.45 / 48.8	2.16 / 39.3
T3	Claude Sonnet 4.6	3.34 / 83.8	3.05 / 72.3	2.60 / 53.2	3.15 / 75.4	3.02 / 73.1	2.57 / 53.9
	GLM 5	3.06 / 74.8	2.83 / 64.8	2.42 / 47.7	2.86 / 66.3	2.65 / 58.1	2.34 / 43.7
	Kimi K2.5	2.94 / 67.4	2.56 / 51.5	2.20 / 38.7	2.65 / 54.9	2.63 / 56.7	2.17 / 36.2
T4	LLM4Decompile	1.79 / 15.2	1.69 / 8.3	1.56 / 1.6	1.63 / 4.1	1.28 / 0.8	1.27 / 0.3
	SK2Decompile	1.54 / 7.3	1.44 / 5.6	1.35 / 0.8	1.45 / 4.4	1.12 / 0.3	1.10 / 0.3
	Claude Sonnet 4.6	2.46 / 43.5	2.34 / 36.1	2.12 / 22.4	2.33 / 34.2	2.21 / 34.2	1.98 / 19.1
	GLM 5	2.29 / 35.3	2.19 / 27.5	2.02 / 17.9	2.19 / 27.6	1.94 / 18.8	1.87 / 13.1
	Kimi K2.5	2.25 / 33.0	2.07 / 22.1	1.97 / 14.3	2.06 / 20.9	1.94 / 19.1	1.79 / 10.6

F Additional Robustness Results

This section repeats the evaluation on subsets of the benchmark and under alternative settings. All results in this section use Claude Sonnet 4.6 as the judge.

F.1 Per-Topic Results

Tables 14–17 report mislead rate by topic and perturbation setting for each task, averaged over all systems. Figure 3b summarizes the P_{comb} results across tasks. Under P_{comb}, mislead rate ranges from 22.9% (Greek Mythology) to 56.6% (MS Security Defender) on T1, from 3.5% (Greek Mythology) to 14.9% (Chess) on T2, from 28.1% (Greek Mythology) to 67.7% (MS Security Defender) on T3, and from 41.7% (Greek Mythology) to 62.4% (MS Security Defender) on T4.

Table 14: T1 function-name recovery: mislead rate (%) by topic $\times$ perturbation vector (averaged over all systems, with Claude Sonnet 4.6 as the judge).

Topic	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
MS Security Defender	35.4	30.8	30.8	49.9	56.6
Unix Commands	17.5	30.6	23.7	27.5	29.5
Quantum Physics	25.5	37.8	34.1	38.6	47.1
Genetics & DNA	20.9	31.5	32.1	26.1	41.8
Chess	26.7	39.9	34.5	36.5	46.1
Greek Mythology	6.5	14.2	13.7	18.3	22.9
Weather	20.0	28.2	25.0	30.0	39.0
Pokemon	21.4	28.5	24.7	25.0	34.1
Video Game	32.8	45.0	45.0	34.3	41.5
Cocktails	7.3	17.7	12.7	27.6	30.7
Mean	21.4	30.4	27.0	32.0	38.9

Table 15: T2 variable-name recovery: mislead rate (%) by topic × perturbation vector (averaged over all systems, with Claude Sonnet 4.6 as the judge).

Topic	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
MS Security Defender	10.9	4.7	5.1	7.1	10.9
Unix Commands	11.7	5.7	4.7	6.2	8.3
Quantum Physics	12.2	5.8	5.1	4.6	9.3
Genetics & DNA	13.3	4.1	5.5	4.0	10.0
Chess	16.1	11.6	9.2	12.6	14.9
Greek Mythology	4.8	0.7	0.6	1.9	3.5
Weather	8.6	3.3	2.9	5.4	7.8
Pokemon	9.2	5.3	4.3	5.9	7.2
Video Game	11.4	11.2	9.9	12.9	11.7
Cocktails	5.8	1.4	0.8	6.6	6.0
Mean	10.4	5.4	4.7	6.9	9.0

Table 16: T3 function summarization: mislead rate (%) by topic × perturbation vector (averaged over all systems, with Claude Sonnet 4.6 as the judge).

Topic	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
MS Security Defender	48.2	57.0	35.1	48.8	67.7
Unix Commands	22.0	51.9	16.3	32.4	44.6
Quantum Physics	25.5	51.4	25.3	32.7	52.0
Genetics & DNA	20.9	41.2	28.6	15.9	43.6
Chess	31.2	53.5	28.4	34.7	53.3
Greek Mythology	8.8	28.9	2.9	19.9	28.1
Weather	18.9	39.0	12.5	26.5	42.1
Pokemon	19.7	37.5	13.7	23.3	33.9
Video Game	39.2	58.7	38.4	33.5	48.7
Cocktails	6.9	26.2	3.5	23.0	31.0
Mean	24.1	44.5	19.2	30.3	44.5

Table 17: T4 decompilation-to-source: mislead rate (%) by topic × perturbation vector (averaged over all systems, with Claude Sonnet 4.6 as the judge).

Topic	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
MS Security Defender	32.3	44.5	32.0	62.3	62.4
Unix Commands	18.3	57.7	27.3	55.3	53.0
Quantum Physics	23.6	62.7	36.1	60.6	60.8
Genetics & DNA	19.9	59.9	54.0	28.9	54.4
Chess	25.9	63.7	39.3	56.1	58.8
Greek Mythology	11.9	55.0	21.7	43.1	41.7
Weather	20.0	63.2	29.0	53.8	55.5
Pokemon	22.3	55.7	29.0	50.6	51.5
Video Game	33.6	70.3	48.1	56.7	57.8
Cocktails	11.4	53.1	23.0	55.4	50.7
Mean	21.9	58.6	31.4	54.8	54.7

F.2 Conditional Results on Correctly Recovered Functions

Table 18 extends Figure 4 to every perturbation setting. For each system, the subset contains the functions whose original prediction receives a semantic-accuracy score of at least 4, and mislead rate is recomputed on their perturbed variants. The subsets for LLM4Decompile and SK2Decompile cover only 0.3% of their predictions, so their conditional rates rest on very few samples.

Table 18: Mislead rate (%) on the functions each system recovers correctly on the original input (accuracy score ≥ 4), with Claude Sonnet 4.6 as the judge. Subset is the percentage of the system's predictions that meet this condition.

Task	System	Subset	P_{var}	P_{func}	P_{type}	P_{str}	P_{comb}
T1	SymGen	4.5%	8.9	67.6	38.7	7.6	69.3
	DeGPT	9.5%	8.0	6.7	6.7	17.9	31.2
	Claude Opus 4.6	18.9%	0.0	3.0	0.1	0.1	1.0
	Claude Sonnet 4.6	14.2%	0.0	1.7	0.3	0.0	0.3
	Claude Haiku 4.5	9.5%	2.1	3.6	3.6	4.8	18.3
	GLM 5	10.0%	1.4	2.8	1.4	0.6	3.8
	Kimi K2.5	10.9%	0.6	1.1	1.7	1.7	4.4
	DeepSeek v3.2	9.3%	1.7	6.2	3.2	4.9	13.5
	Qwen3-Coder-Next	6.6%	7.6	10.3	7.6	41.8	45.8
T2	GenNm	1.4%	4.5	7.6	3.0	3.0	3.8
	ReSym	11.2%	15.7	13.5	6.1	11.8	18.7
	VarBERT	14.3%	4.1	3.5	2.7	2.8	5.3
	Claude Opus 4.6	38.5%	8.2	1.1	0.1	0.1	4.5
	Claude Sonnet 4.6	34.8%	9.9	1.0	0.2	0.3	5.0
	Claude Haiku 4.5	26.4%	8.8	2.1	2.3	2.2	6.6
	GLM 5	27.8%	8.5	2.3	1.2	1.4	7.8
	Kimi K2.5	25.8%	11.2	2.6	1.3	0.6	7.8
	DeepSeek v3.2	24.1%	8.8	3.9	2.6	2.9	9.5
	Qwen3-Coder-Next	21.7%	6.9	3.9	3.7	6.5	14.9
T3	Claude Opus 4.6	41.0%	2.6	13.8	0.6	1.0	10.0
	Claude Sonnet 4.6	34.5%	3.3	15.0	1.9	0.4	11.8
	Claude Haiku 4.5	22.1%	10.4	22.8	7.0	10.6	39.4
	GLM 5	24.1%	4.3	17.9	2.9	2.3	14.8
	Kimi K2.5	26.3%	8.1	20.7	5.6	3.8	19.3
	DeepSeek v3.2	20.6%	6.3	19.8	4.0	11.5	28.9
	Qwen3-Coder-Next	17.0%	13.5	24.1	13.3	65.8	67.6
T4	LLM4Decompile	0.3%	20.0	60.0	40.0	60.0	100.0
	SK2Decompile	0.3%	0.0	81.3	0.0	62.5	75.0
	Claude Opus 4.6	9.3%	0.0	17.2	0.2	2.2	11.8
	Claude Sonnet 4.6	7.8%	0.0	12.8	0.3	1.3	10.5
	Claude Haiku 4.5	2.5%	2.4	34.4	0.8	48.0	46.4
	GLM 5	3.0%	0.7	10.0	2.0	14.0	18.7
	Kimi K2.5	3.1%	0.6	19.4	1.9	16.1	22.6
	DeepSeek v3.2	2.1%	1.0	21.9	4.8	41.9	34.3
	Qwen3-Coder-Next	1.0%	2.0	12.0	8.0	84.0	60.0

F.3 Optimization-Level Robustness

We rebuild the benchmark functions at O0 and repeat the evaluation with the same configuration. Table 19 reports the results. Under P_{comb}, the average mislead rates are 39.0%, 8.6%, 44.3%, and 57.2% for T1–T4, compared with 38.9%, 9.0%, 44.5%, and 54.7% at O2. Across the 35 system–task pairs, the P_{comb} mislead rates at O0 and O2 have a Spearman's ρ of 0.991.

Table 19: Accuracy and mislead rate (%) with the benchmark rebuilt at O0 and Claude Sonnet 4.6 as the judge, averaged over the 10 topics.

Task	System	Orig.	P_{var}		P_{func}		P_{type}		P_{str}		P_{comb}	
		Acc	Acc	Mis	Acc	Mis	Acc	Mis	Acc	Mis	Acc	Mis
T1	SymGen	4.4	3.0	20.5	0.9	78.9	2.2	60.4	1.7	18.8	0.7	77.5
	DeGPT	8.7	5.5	36.1	5.3	32.5	6.3	41.6	4.5	62.5	2.1	59.0
	Claude Opus 4.6	17.4	14.1	8.8	14.3	22.0	15.2	7.1	11.8	5.4	10.6	12.2
	Claude Sonnet 4.6	14.0	10.2	12.8	10.4	15.4	11.5	13.2	9.7	13.9	7.5	14.2
	Claude Haiku 4.5	9.2	7.1	22.9	6.7	23.1	7.4	26.1	5.6	33.7	4.0	40.4
	GLM 5	9.1	8.0	12.2	7.7	17.5	8.2	16.3	7.2	14.8	5.7	15.7
	Kimi K2.5	10.1	7.8	17.2	7.4	20.3	8.4	18.5	8.0	18.7	5.7	20.8
	DeepSeek v3.2	8.1	7.0	21.0	5.8	27.2	7.1	24.9	5.4	34.2	4.1	36.8
	Qwen3-Coder-Next	7.4	5.6	38.9	4.3	36.6	5.2	42.1	2.5	83.3	1.9	74.1
T2	GenNm	1.3	0.1	2.1	1.3	3.0	0.7	2.6	0.4	2.3	0.7	3.8
	ReSym	11.0	6.1	27.8	9.1	11.9	11.1	11.3	6.1	25.2	5.7	25.1
	VarBERT	14.3	13.6	2.3	7.9	3.4	6.6	3.6	7.6	6.6	6.0	4.2
	Claude Opus 4.6	38.0	39.0	7.7	36.0	3.1	38.3	0.6	36.2	0.9	36.0	3.9
	Claude Sonnet 4.6	34.9	35.1	8.3	31.3	3.0	33.5	1.1	32.2	1.0	29.8	4.5
	Claude Haiku 4.5	27.0	23.4	9.3	23.4	4.7	24.2	4.5	21.7	3.5	17.6	7.1
	GLM 5	26.6	25.1	9.2	23.4	3.4	25.5	1.9	23.5	1.4	20.9	5.6
	Kimi K2.5	25.4	21.1	11.9	22.5	4.7	23.5	3.4	24.5	1.6	17.9	5.8
	DeepSeek v3.2	25.8	24.8	9.6	21.7	8.6	23.7	7.3	21.9	6.4	19.5	9.8
	Qwen3-Coder-Next	24.4	21.9	12.4	20.7	7.3	21.5	8.0	17.9	14.0	14.6	16.0
T3	Claude Opus 4.6	39.8	32.3	13.1	26.9	37.0	36.4	4.6	32.7	7.2	22.4	23.3
	Claude Sonnet 4.6	34.0	27.5	14.6	22.5	39.8	30.7	10.0	26.8	11.2	17.0	28.1
	Claude Haiku 4.5	22.3	13.0	29.7	10.8	47.3	15.2	26.1	8.0	35.9	4.6	58.8
	GLM 5	23.1	17.7	14.9	13.2	40.9	18.5	12.7	14.3	17.0	10.2	32.0
	Kimi K2.5	26.1	18.6	26.5	14.6	47.1	20.8	22.8	17.4	17.5	10.6	37.2
	DeepSeek v3.2	21.4	16.3	21.6	11.3	44.7	17.5	18.2	11.2	33.9	7.3	48.8
	Qwen3-Coder-Next	15.8	9.8	39.1	6.9	50.8	10.9	37.2	2.5	84.8	2.2	81.6
T4	LLM4Decompile	0.4	0.4	46.5	0.0	84.1	0.3	77.8	0.1	83.2	0.0	95.4
	SK2Decompile	0.3	0.1	2.4	0.0	72.9	0.1	20.0	0.0	79.6	0.0	79.9
	Claude Opus 4.6	9.0	6.4	8.9	4.3	55.7	6.0	7.0	3.6	28.5	2.5	34.7
	Claude Sonnet 4.6	7.1	5.3	13.6	2.7	44.2	4.7	13.1	2.9	24.2	1.6	28.2
	Claude Haiku 4.5	2.5	1.3	22.5	0.7	57.4	1.4	29.2	0.3	69.2	0.2	60.8
	GLM 5	2.7	2.1	12.6	1.2	43.2	2.0	18.3	0.6	36.2	0.5	32.8
	Kimi K2.5	3.4	2.0	20.4	1.0	54.5	2.1	28.4	1.2	41.1	0.6	46.8
	DeepSeek v3.2	1.9	1.3	26.6	0.6	56.6	1.3	38.7	0.5	63.4	0.3	62.3
	Qwen3-Coder-Next	1.6	1.1	39.2	0.8	55.4	0.9	46.9	0.0	85.6	0.0	73.7

F.4 Source-Level versus Pseudocode-Level Perturbation

Appendix B.1 checks whether source-level perturbations reach the pseudocode. Here we compare the evaluation results: we apply P_{comb} to the source code of the same functions and give the perturbed source code to each system, with the same configuration. Table 20 reports the results. Under P_{comb}, the average mislead rates on source code are 46.2%, 10.6%, 59.1%, and 71.0% for T1–T4, compared with 38.9%, 9.0%, 44.5%, and 54.7% on pseudocode.

Table 20: Accuracy and mislead rate (%) under P_{comb} applied to the pseudocode (as in the main evaluation) and to the source code, with Claude Sonnet 4.6 as the judge, averaged over the 10 topics.

Task	System	Pseudocode Acc	Pseudocode Mis	Source code Acc	Source code Mis
T1	SymGen	0.7	77.0	0.9	83.4
	DeGPT	2.3	59.7	8.0	67.2
	Claude Opus 4.6	10.9	11.2	22.8	19.7
	Claude Sonnet 4.6	7.7	14.4	16.7	21.7
	Claude Haiku 4.5	3.6	39.5	10.4	47.5
	GLM 5	6.8	14.7	13.5	23.9
	Kimi K2.5	6.2	20.4	14.0	29.3
	DeepSeek v3.2	4.3	37.6	11.7	43.8
	Qwen3-Coder-Next	1.4	75.9	5.5	79.2
T2	GenNm	0.7	3.6	1.9	4.3
	ReSym	5.7	25.2	12.2	28.3
	VarBERT	6.1	4.3	12.4	6.4
	Claude Opus 4.6	35.8	3.6	62.1	1.2
	Claude Sonnet 4.6	29.5	3.9	57.4	2.2
	Claude Haiku 4.5	17.5	7.5	39.2	10.4
	GLM 5	21.5	4.7	43.5	4.8
	Kimi K2.5	17.7	5.5	34.5	7.9
	DeepSeek v3.2	16.9	10.0	41.0	14.4
	Qwen3-Coder-Next	8.8	21.4	34.5	26.5
T3	Claude Opus 4.6	24.7	22.9	33.9	39.9
	Claude Sonnet 4.6	16.5	28.2	26.4	48.8
	Claude Haiku 4.5	4.5	57.2	9.7	70.4
	GLM 5	10.7	29.7	18.8	49.2
	Kimi K2.5	11.0	38.7	19.3	53.1
	DeepSeek v3.2	7.0	46.9	13.3	65.6
	Qwen3-Coder-Next	1.6	87.8	5.4	86.8
T4	LLM4Decompile	0.0	90.6	0.1	97.5
	SK2Decompile	0.0	78.5	0.0	87.1
	Claude Opus 4.6	2.9	29.1	9.6	52.4
	Claude Sonnet 4.6	1.9	33.3	6.0	49.7
	Claude Haiku 4.5	0.2	60.8	1.4	76.0
	GLM 5	0.5	30.5	4.3	54.2
	Kimi K2.5	1.1	45.0	4.0	64.5
	DeepSeek v3.2	0.5	42.2	2.6	74.8
	Qwen3-Coder-Next	0.0	81.9	1.3	82.6