Train in Vain: Functionality-Preserving Poisoning to Prevent Unauthorized Use of Code Datasets

Yuan Xiao[1*] **Jiaming Wang**[1*] **Yuchen Chen**[1*] **Wei Song**[2] **Jun Sun**[3]
Shiqing Ma[4] **Yanzhou Mu**[1] **Juan Zhai**[4] **Chunrong Fang**[1†] **Jin Song Dong**[5] **Zhenyu Chen**[1†]
[1]Nanjing University [2]University of New South Wales [3]Singapore Management University
[4]University of Massachusetts Amherst [5]National University of Singapore
yuan.xiao@smail.nju.edu.cn, 231220072@smail.nju.edu.cn, yuc.chen@smail.nju.edu.cn
wei.song1@unsw.edu.au, junsun@smu.edu.sg, shiqingma@umass.edu
602022320006@smail.nju.edu.cn, juanzhai@umass.edu, fangchunrong@nju.edu.cn
dcsdjs@nus.edu.sg, zychen@nju.edu.cn

Abstract

The widespread availability of large-scale code datasets has accelerated the development of code large language models (CodeLLMs), raising concerns about unauthorized dataset usage. Dataset poisoning offers a proactive defense by reducing the utility of such unauthorized training. However, existing poisoning methods often require full-dataset poisoning and introduce transformations that break code compilability. In this paper, we introduce FUNPOISON, a functionality-preserving poisoning approach that injects short, compilable weak-use fragments into executed code paths. FUNPOISON leverages reusable statement-level templates with automatic repair and conservative safety checking to ensure side-effect freedom, while a type-aware synthesis module preserves type correctness, suppresses static-analysis warnings, and improves stealth. Extensive experiments across multiple CodeLLMs and code-generation benchmarks show that FUNPOISON achieves effective poisoning by contaminating only 10% of the dataset, while maintaining 100% compilability and functional correctness. FUNPOISON also remains robust against advanced code sanitization techniques, including detection, purification, rewriting, static-analysis, and formatting defenses.

1 Introduction

Code large language models (CodeLLMs) (GitHub, 2022; Beijing Guixin Technology, 2022; Amazon Web Services, 2023) have achieved strong performance across a wide range of code understanding and generation tasks. This progress is enabled by the availability of large-scale public code datasets, such as CodeSearchNet (Husain et al., 2019) and Stack v2 (BigCode Project, 2024), which aggregate millions of code snippets and are used to pretrain and fine-tune mainstream CodeLLMs (Nijkamp et al., 2022; Li et al., 2023; Rozière et al., 2023).

However, the widespread use of large-scale code datasets in training pipelines raises compliance and copyright concerns, as many code contributors neither expect nor consent to their code being used for model training and fine-tuning (News, 2022; Legal.io, 2024). Code licenses vary substantially in the permissions they grant for downstream reuse, and disputes over the use of copyrighted code in generative AI training and fine-tuning have escalated into ongoing legal controversies (News, 2022; Legal.io, 2024; Firm, 2024). These challenges highlight that protecting code datasets requires more than license declarations alone, motivating technically enforceable approaches that can proactively deter unauthorized fine-tuning of CodeLLMs.

Dataset poisoning (Sun et al., 2022) has emerged as a proactive protection mechanism for safeguarding code datasets against unauthorized training (Sun et al., 2022). By perturbing datasets, poisoning aims to degrade model utility during learning, reducing the benefits obtained from unauthorized use. This deterrence-oriented property makes poisoning particularly appealing, as it does not rely on post hoc attribution like code watermarking (Sun et al., 2023; Xiao et al., 2025; Chen et al., 2026)

[*]Equal contribution. **Yuan Xiao** led the overall project, defined the research problem, technical direction, and experimental agenda, carried out the key final implementation and method convergence, and took primary responsibility for paper writing, revision, and submission; **Jiaming Wang** contributed to early-stage exploration, implementation, evaluation runs, and dynamic analysis under Yuan Xiao's guidance; **Yuchen Chen** contributed substantially to experiment construction, robustness experiments, paper revision, and most rebuttal-stage experiments.

[†]Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 31284–31303

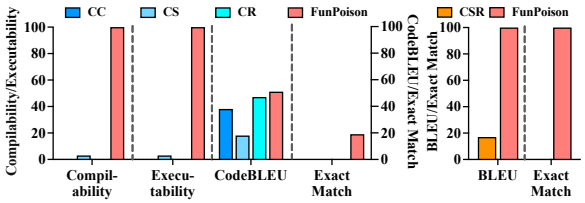

Figure 1: Quality evaluation of FUNPOISON and CoProtector on fully poisoned (100%) test datasets (984 inputs, obtained via six-fold duplication of the Java subset of HumanEval-X with 164 tasks). Left: code quality; Right: comment quality.

or legal enforcement (News, 2022; Legal.io, 2024; Firm, 2024), which are often costly, delayed, or ineffective.

CoProtector (Sun et al., 2022) is the first, and to date the only untargeted poisoning framework for code datasets. It aims to deter unauthorized use by degrading overall model performance. CoProtector applies four transformations: code corruption (CC), injecting syntactically invalid code; code splicing (CS), inserting code fragments from other programs; code renaming (CR), obfuscating identifiers; and comment semantic reversal (CSR), altering comment semantics without modifying executable code. However, CoProtector suffers from fundamental limitations that undermine its practicality. Its transformations are either functionally destructive (CC, CS, CR), leading to near-zero compilability, or semantically shallow (CSR), degrading the comment quality (Fig.1). Figure 1 compares intrinsic quality preservation under fully poisoned samples; the partial-poisoning effectiveness comparison is reported in RQ1. As shown in Fig. 2, CoProtector achieves deterrence only in the extreme case of full-dataset poisoning; under partial poisoning, fine-tuned models still obtain clear performance gains over the base model. Together, the degradation of code quality and reliance on unrealistic poisoning assumptions fundamentally limit the practicality of existing approaches, highlighting the lack of functionality-preserving poisoning methods for code datasets.

In this paper, we introduce FUNPOISON, a **fun**ctionality-preserving **poison**ing framework that deters unauthorized fine-tuning of Code LLMs without sacrificing code usability (Fig. 3). FUNPOISON achieves this by injecting short, execution-inert code fragments into executed paths. It constructs a reusable template pool from real-world statement-level code, repairs fragments for independent compilability, and applies conservative safety filtering to eliminate side-effect-prone snippets. During deployment, templates are injected only at effect-free sites and augmented with type-aware weak-use statements to prevent removal by static analysis. Together, these designs enforce ***compilability***, ***functionality preservation***, and ***practical persistence***, which are essential for effective poisoning under realistic partial-dataset settings, as illustrated by the end-to-end pipeline in Figure 2. Consistent with this design, experiments show that FUNPOISON achieves strong poisoning with only a 10% injection ratio, suppressing fine-tuning gains while maintaining 100% compilation success with negligible runtime overhead. Additional mechanism and ablation studies show that degradation is driven by execution-path supervision rather than superficial template exposure. FUNPOISON also remains effective under the evaluated static analysis, removal, rewriting, formatting, and adaptive-detection settings.

In summary, we make three major contributions:

- We propose FUNPOISON, a poisoning-based protection framework for code datasets that deters unauthorized fine-tuning while preserving normal code usability.
- FUNPOISON introduces a functionality-preserving poisoning mechanism that injects weak-use code fragments into executed paths, ensuring training-time influence while preserving program semantics, compilability, and runtime behavior.
- We provide a mechanism analysis and a controlled DeadBranchInsertion ablation showing that execution-path supervision, rather than template exposure alone, explains the degradation effect.
- Experiments across model scales, code-generation benchmarks, and adaptive defenses show that FUNPOISON is effective under partial poisoning (10%) while maintaining 100% compilability and functional correctness. Our code: (Xiao et al., 2026).

2 Threat Model

Dataset poisoning is a proactive defense against unauthorized training or fine-tuning of CodeLLMs on protected datasets (Sun et al., 2022). We consider a dataset owner who releases usable code artifacts for ordinary development use, but does not authorize large-scale model adaptation. The attacker collects the released data and fine-tunes a

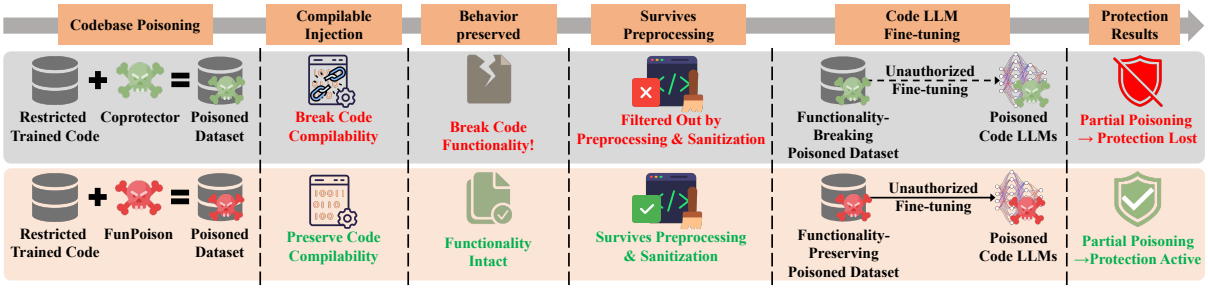

Figure 2: End-to-end comparison of poisoning pipeline of CoProtector and FUNPOISON.

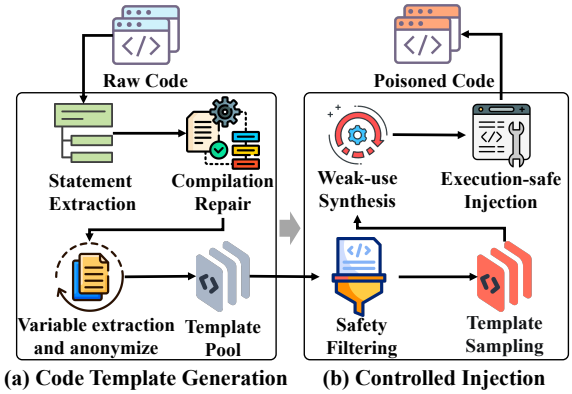

Figure 3: Overview of FUNPOISON.

pretrained CodeLLM to obtain downstream performance gains without authorization.

The attacker controls the training pipeline, including model choice, optimization, data preprocessing, formatting, static analysis, purification, and LLM-based rewriting. The attacker may also know that FUNPOISON is used and may build signature rules or supervised detectors. However, the attacker does not have access to the clean version reserved for authorized training users, cannot manually inspect web-scale data, and must preserve benign code while maintaining an acceptable false-positive rate.

A poisoning defense is effective when unauthorized fine-tuning fails to yield meaningful gains over the base model. Benign users can still compile, run, test, and integrate the released code because FUNPOISON preserves observable program behavior. Authorized training users can be given clean data through licensing, access control, or provenance-verified releases; such governance mechanisms are complementary to the technical defense.

3 Methodology

FUNPOISON is a functionality-preserving poisoning framework designed to deter unauthorized fine-tuning of Code LLMs without sacrificing code usability (Fig. 3). Its design is guided by three guarantees: ***compilability***, ***functionality preservation***, and ***practical persistence***. To satisfy these guarantees, FUNPOISON synthesizes execution-inert, compilable templates and injects them into executed code paths under strict safety constraints.

Algorithm 1 FunPoison

INPUT: code dataset $\mathcal{D}$, poisoning rate $\rho \in (0, 1]$
OUTPUT: poisoned dataset $\mathcal{D}'$

function FUNPOISON($\mathcal{D}$)
  $\mathcal{F} \leftarrow \emptyset$ ▷ template pool
  for all $c \in \mathcal{D}$ **do**
    $S \leftarrow$ EXTRACTSTATEMENTS(c)
    for all $s \in S$ **do**
      $s' \leftarrow$ REPAIRTOCOMPILABLE(s)
      if $s' = \emptyset$ **then**
        continue
      end if
      $t \leftarrow$ ANONYMIZEVARIABLE(s')
      $\mathcal{F} \leftarrow \mathcal{F} \cup \{t\}$
    end for
  end for
  $\mathcal{D}_{\text{poi}} \leftarrow$ SAMPLESUBSET($\mathcal{D}, \rho$)
  $\mathcal{D}' \leftarrow \mathcal{D}' / \mathcal{D}_{\text{poi}}$
  for all $c \in \mathcal{D}_{\text{poi}}$ **do**
    $\mathcal{F}' \leftarrow$ SAFETYFILTER($\mathcal{F}$)
    $L \leftarrow$ SELECTINSERTIONSITES(c)
    $T \leftarrow$ SAMPLETEMPLATES($\mathcal{F}'$)
    for all $t \in T$ **do**
      $w \leftarrow$ SYNTHESIZEWEAKUSE(t)
      $c \leftarrow$ INJECT(c, t, w, L)
    end for
    $\mathcal{D}' \leftarrow \mathcal{D}' \cup \{c\}$
  end for
end function

3.1 Code Template Generation

This module constructs the template pool that underpins FUNPOISON, enforcing the guarantees of **compilability** and **functionality preservation** required for downstream poisoning. Effective and stealthy poisoning requires code fragments that can be reused across projects while remaining safe to transplant. We therefore draw templates from large-scale real-world code corpora, since poisoning based on narrow or repetitive patterns is more likely to be brittle or easily sanitized. However, raw snippets from such corpora are rarely reusable in their

original form, as they are often incomplete, context-dependent, or unsafe when relocated. To address this, FUNPOISON adopts a compilation-driven and name-aware pipeline that refines real-world code into reusable templates, as shown in lines 2-13 of Alg. 1. Concretely, template generation is organized into three steps: *(i) Statement Extraction*, identifying executable statement-level fragments; *(ii) Compilation Repair*, converting them into independently compilable units; and *(iii) Name Extraction and Conflict-Aware Reuse*, tracking identifier metadata and normalizing or renaming names when needed.

Statement Extraction. As the first step, we extract statement-level fragments from executable code outside type declarations (Alg. 1, line 4). Statement-level units strike a balance between expressiveness and portability: they capture diverse real-world usage patterns while remaining lightweight enough to be reused across different contexts. Many extracted statements are not self-contained, as they may reference implicit context such as surrounding declarations or imports. Discarding such fragments would significantly reduce the coverage and diversity of the template pool. Instead, we retain these candidates and normalize them into standalone statement fragments by removing irrelevant type blocks and comments, and partitioning the remaining code into individual executable statements. This step yields a broad and diverse set of statement-level candidates that serve as the input to subsequent compilation repair and safety enforcement stages.

Compilation repair. The goal of compilation repair is to maximize template coverage while guaranteeing independent compilability, which is essential for safe large-scale injection. Statement-level fragments extracted from real-world code are often incomplete or context-dependent (e.g., missing imports, undeclared types, or bare allocations), and discarding them would substantially reduce diversity. We therefore adopt a compilation-driven repair strategy that reconstructs only the minimal context required for successful compilation. Concretely, REPAIRTOCOMPILE(Alg. 1, line 6) resolves all referenced types and normalizes incomplete constructs. Referenced classes are analyzed to distinguish standard library types from user-defined ones: JDK classes are materialized as explicit imports, while non-JDK types are replaced with lightweight same-file stubs to eliminate missing dependencies.[1] Fully qualified names are simplified to short forms and redundant stubs are removed to avoid naming conflicts.[2] Bare allocations (e.g., `new T(args);`) are rewritten as assignments with fresh local variables to ensure syntactic completeness.[3] The repaired fragment, together with its synthesized imports, declarations, and stubs, is wrapped into a minimal compilation unit and validated via JAVAC, retaining only fragments that compile successfully. This process consistently transforms incomplete snippets into stable, self-contained templates for downstream name handling, filtering, and controlled injection.

Name Extraction and Conflict-Aware Reuse. After REPAIRTOCOMPILE, each candidate statement is represented as a set s' that bundles the repaired snippet with its required imports, synthesized preamble, and lightweight stubs, yielding a pool of concise and independently compilable templates suitable for reuse. The ANONYMIZEVARIABLE routine (Alg. 1, line 10) extracts identifiers, method names, and class names, and records both the original names and placeholder-normalized variants. In the default injection pipeline, non-conflicting names are preserved to maintain natural code style, while local variables that collide with identifiers in the host scope are renamed to fresh names during conflict resolution. Finally, the repaired snippet, its auxiliary context, and the associated name metadata are consolidated into a normalized record t and added to the global template set $\mathcal{F}$ (lines 11), which serves as the foundation for subsequent controlled injection.

The template pool can be extracted from the protected corpus or from an external corpus. In our experiments, we randomly sample templates from a large protected corpus, which improves syntactic compatibility while avoiding repeated reuse of any small set of source fragments. Some original identifier names may be preserved when they are available and non-conflicting, but the retained context is constrained by statement-level extraction

[1] Stubs contain empty method bodies and no side effects.

[2] For example, occurrences of `java.util.List` are rewritten as `List` with an explicit import added. Any synthesized stub that would shadow or duplicate an imported standard-library type is removed, ensuring that each type name in the compilation unit has a unique and unambiguous definition.

[3] A bare allocation refers to a standalone object construction without an assignment target (e.g., `new T();`), which is often flagged as useless or removed by static analysis and preprocessing. Rewriting it as an assignment (e.g., `T tmp = new T();`) preserves compilability and enables subsequent weak-use synthesis without affecting program behavior.

and minimal compilable repair: templates typically contain only sparse local names, synthesized declarations, and lightweight stubs rather than complete algorithms or surrounding control-flow logic. If desired, a deployment can further reduce these local lexical traces by using an external template source or applying stronger name normalization before release.

3.2 Controlled Injection

This module enforces ***functionality preservation*** and ***stealthy persistence*** via controlled injection, comprising *(i) safety filtering*, *(ii) weak-use synthesis*, and *(iii) execution-safe site selection*.

Safety Filtering. In this stage (Alg.1, line17), we enforce behavioral safety under a conservative policy, addressing the risk that seemingly harmless fragments may become unsafe when relocated. We therefore prioritize aggressive pruning, removing any template that could affect compilability, execution stability, or portability. To achieve this, we apply a two-tier safety filtering framework (Table 7 in the Appendix A.1.1) that jointly addresses semantic and operational risks. The first tier consists of *conceptual rules* based on semantic reasoning, which exclude patterns that may appear syntactically valid but are unsafe when relocated across projects, such as control-flow disruptions, reflective dependencies, or shared-state interactions. The second tier applies *programmatic rules* using lightweight static analysis to remove fragments exhibiting concrete side effects, including I/O, concurrency, process control, and non-local state mutations. Together, these filters ensure that retained templates are execution-inert and portable, enabling safe controlled injection.

Weak-use Synthesis. Conceptually, weak-use synthesis ensures that injected fragments survive compilation and preprocessing, preserving poisoning signals without altering observable behavior. The weak-use pool balances three requirements—diversity, behavioral neutrality, and conciseness—to avoid repetitive patterns prone to detection, preserve program semantics, and minimize interference with compilation, thereby improving the robustness and stealth of injected fragments. To address this, we synthesize type-driven *weak-use* statements (Alg.1, line 21) that semantically consume declared variables without altering program behavior. For each variable, a weak-use expression is sampled from a curated pool (Table 8 in the Appendix A.1.2) covering common types, including primitives, collections, maps, optionals, and generic objects. The pool is designed to ensure type correctness and portability, restrict usage to identity or metadata queries (e.g., boxing, identity hashing, guarded size checks), and exclude I/O, concurrency, or global-state mutations. Multiple patterns are provided for each variable type to avoid repetitive structures. While weak-use synthesis ensures that injected fragments are retained during compilation and preprocessing, safe deployment further depends on where these fragments are placed within the host program.

Execution-safe Site Selection. The goal of site selection is to ensure that injected templates remain compilable and execution-inert after deployment. To this end, we restrict injection to syntactically stable and semantically inert seams within method bodies (Alg. 1, lines 18, 22), avoiding positions where even benign code could alter control flow or observable behavior. Concretely, we scan each method body while tracking scope structure, and consider a location valid only if the preceding statement is inert, i.e., either (i) a pure declaration or (ii) a side-effect–free expression. We exclude anchors adjacent to control-transfer statements (e.g., `return`, `throw`, `break`, `continue`), near method boundaries, or involving operations with observable effects such as I/O, process control, container mutation, or non-local assignments.[4] If no valid site exists, the file is skipped; otherwise, m templates are uniformly assigned to valid sites. At injection time, we resolve identifier conflicts to preserve local scope correctness: template's variables that collide with host identifiers are systematically renamed using fresh names, while non-conflicting identifiers are preserved. The template is then integrated by harmonizing imports, preambles, and identifiers with the host context, ensuring successful compilation and preserving execution behavior.

4 Evaluation

Our evaluation addresses four research questions:

- **RQ1.** How effective is FUNPOISON at degrading model performance on code generation?
- **RQ2.** How well does FUNPOISON preserve code functionality and quality?
- **RQ3.** How robust is FUNPOISON?

[4] Comments are ignored and brace depth is tracked to ensure scope correctness.

- **RQ4.** How does the poisoning effect of FUNPOISON vary across settings?

4.1 Experiment Setup

Datasets, Models, and Training. We sample 100K Java functions from CodeSearchNet (CSN) (Husain et al., 2019). Experiments are conducted on DeepSeek-Coder (1.3B and 6.7B) (Guo et al., 2024) and StarCoderBase (1B) (Li et al., 2023); we further evaluate CodeLlama-7B and CodeLlama-7B-Instruct (Rozière et al., 2023) to test larger and instruction-tuned settings. We apply LoRA-based fine-tuning for DeepSeek-Coder-6.7B and full-parameter fine-tuning for other models. The primary evaluation uses HumanEval-X (Zheng et al., 2023). We additionally evaluate MBPP (Austin et al., 2021) to test benchmark transfer within executable code generation. Additional training details are in Appendix A.2.

Baselines. We compare against CoProtector (Sun et al., 2022), the only existing dataset-level poisoning method explicitly proposed for copyright protection of code datasets. We apply its four transformations (CS, CC, CR, and CSR) following the original implementation and settings. We also introduce DeadBranchInsertion as a controlled ablation: it uses the same template pool as FUNPOISON but places templates inside always-false branches, isolating whether degradation comes from execution-path supervision rather than template exposure alone.

Attack Methods. We evaluate the robustness of FUNPOISON against representative attacks, including classical detection and purification methods (Spectral Signature (SS) (Tran et al., 2018) and Activation Clustering (AC) (Chen et al., 2019)), recent code-specific attacks (KillBadCode (Sun et al., 2025) and DeCoMa (Xiao et al., 2025), the only method targeting watermark/trigger removal in code datasets), static analysis via CodeQL (GitHub Inc., 2025), formatter-based normalization using `clang-format` (LLVM Project, 2023)(to better match the original formatting style of CSN), LLM-based rewriting with CodeLlama-7B-Instruct (Rozière et al., 2023) and GPT-4 (OpenAI, 2023), and an adaptive supervised detector implemented with a CodeBERT classifier (Feng et al., 2020). Additional details are in Appendix A.2.

Evaluation Metrics We use ΔPass@k to evaluate ***poisoning effectiveness***, defined as the difference in Pass@k (Chen et al., 2021) between the fine-tuned and base models on HumanEval-X (Zheng et al., 2023). Pass@k measures the probability that at least one of the top-k generated candidates passes all unit tests, and we report $k \in \{1, 3, 5\}$. Pass@k is evaluated under decoding temperatures $\{0.0, 0.2, 0.4\}$ and poisoning rates $\{1\%, 10\%, 50\%, 100\%\}$. At decoding temperature $T = 0.0$, generation is deterministic; therefore, Pass@1 is equivalent to Pass@3 and Pass@5, and we report only Pass@1. We assess ***execution harmlessness*** via dynamic analysis across performance overhead, coverage, runtime stability, and behavior consistency, using GNU time (Free Software Foundation, 2025), JaCoCo (JaCoCo, 2025), and Python (`difflib`) (Python Software Foundation, 2025b,a), with programs compiled and executed via `javac` and the JVM (Oracle Corporation, 2025) under fixed settings. We also measure ***similarity*** between code before and after injection using Exact Match (Rajpurkar et al., 2016), BLEU (Papineni et al., 2002), and CodeBLEU (Ren et al., 2020). Details are provided in Appendix A.2.1.

5 Evaluation Results

RQ1: How effective is FUNPOISON at degrading model performance on code generation? To evaluate the effectiveness of FUNPOISON (i.e., its ability to degrade the performance of Code LLMs fine-tuned on FUNPOISON-poisoned datasets), we conduct code generation experiments on DeepSeek-Coder-1.3B, varying the poisoning rate from 1% to 100% and testing decoding temperatures of 0.0, 0.2, and 0.4. Fig. 4 shows the results. When the poisoning rate reaches 10%, FUNPOISON already yields a clear negative effect (with ΔPass@1 becoming significantly negative), and this degradation intensifies as the poisoning rate increases (e.g., it is more pronounced at 50%). Notably, the trends are largely consistent across decoding temperatures, indicating stable poisoning effectiveness under different sampling strategies. In contrast, CSR does not significantly reduce ΔPass@1 at any poisoning rate, while CC, CR, and CS only show noticeable impact under 100% poisoning. Results for ΔPass@3/5 are provided in Appendix A.6.

Mechanistic analysis. Although injected fragments are inert at runtime, they are not inert during autoregressive fine-tuning: the model minimizes next-token loss over all tokens in method bodies, including the weak-use fragments embedded in executable regions. Table 1 shows that generated

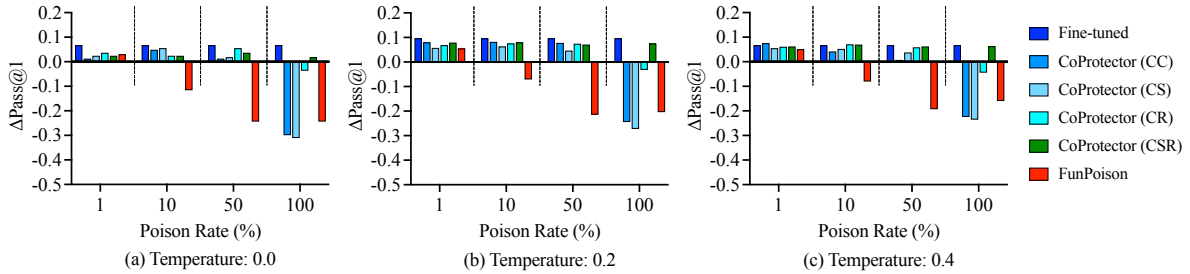

Figure 4: Poisoning effects of full-parameter fine-tuning DeepSeek-Coder-1.3B on datasets poisoned by FUNPOISON and CoProtector.

Table 1: Mechanistic evidence: injected structural patterns strongly co-occur with failed generations. The $10\%, T = 0.0$ row aggregates 820 generations, the 10% all-temperature row aggregates 2,460 generations, and the total aggregates 9,840 generations.

Setting	Weak-use fail/total	Template fail/total
$10\%, T = 0.0$	415 / 435	755 / 785
10%, all T	1295 / 1363	2231 / 2344
All rates/all T	5328 / 5653	8093 / 8810

Table 2: Controlled ablation with the same template pool. DeadBranchInsertion places templates in always-false branches; Pass@1 is reported at $T = 0.0$.

Variant	Poisoning rate	Pass@1
Base	–	0.31
Clean fine-tuned	0%	0.38
FUNPOISON	10%	**0.20**
DeadBranchInsertion	1%	0.37
	10%	0.38
	50%	0.34
	100%	0.35

weak-use and template signatures overwhelmingly co-occur with execution failures. This correlation alone does not establish causality, so we isolate the effect using DeadBranchInsertion, which exposes the model to the same template pool but removes execution-path placement. As shown in Table 2, DeadBranchInsertion at 10% matches clean fine-tuning, while FUNPOISON at the same rate reduces Pass@1 to 0.20. These results support the view that degradation is driven by execution-path supervision and distributional interference, rather than by superficial exposure to template text.

RQ2: How well does FUNPOISON preserve code functionality and quality?

Functionality preservation and semantic similarity. We evaluate whether FUNPOISON preserves executable behavior and semantic fidelity using compilability, Pass@1, and BLEU/CodeBLEU on the Java subset of HumanEval-X (164 tasks, duplicated six times to 984 instances). Since CSR perturbs only comments, we evaluate it with comment-level metrics (BLEU and Exact Match), while code-level baselines are assessed using Pass@1 and BLEU/CodeBLEU. As shown in Fig. 1, CC, CS, and CR severely impair code usability, with compilability and executivity near zero. In contrast, FUNPOISON achieves 100% executivity on all instances. It also attains the highest CodeBLEU among code-level baselines, while CC, CS, and CR exhibit substantial semantic degradation. For comment quality, CSR preserves executability but sharply degrades semantics, whereas FUNPOISON maintains near-perfect similarity. Overall, FUNPOISON consistently preserves executability while maintaining substantially higher semantic fidelity than all CoProtector variants.

Table 3: Dynamic analysis results comparing clean and poisoned code across 984 tasks.

Metric	Clean	FUNPOISON
Compilation success	984/984	984/984
p95 time overhead	–†	2.29% mean (p95: 25%)
p95 memory overhead	–†	0.09% mean (p95: 2.41%)
Coverage (lines)	100%	100%
Execution jitter	8.17%	8.12%
Behavior consistency	Preserved*	Preserved*

†: Clean runs serve as the baseline and thus have no overhead values.
*: "Preserved" means that program outputs, exceptions, and I/O behaviors of poisoned code remain identical to the clean version (functional equivalence).

Execution harmlessness via dynamic analysis. To further ensure that FUNPOISON does not compromise the functionality of host programs, we perform dynamic analysis on 984 Java tasks and compare clean and poisoned code across multiple runtime dimensions (Table 3). The results show that all poisoned programs run correctly and incur negligible overhead: the mean p95 latency increase is only 2.29%, with memory overhead below 0.1%. Line coverage remains identical (100%), indicating that the injected fragments do not alter execution paths. Stability analysis shows nearly identical jitter (8.17% vs. 8.12%), and behavior consistency is fully preserved, with no differences in outputs,

Table 4: Performance of LLMs in rewriting 500 poisoned code samples.

	CodeLlama			GPT-4		
	ACC	BLEU*	Time (s)	ACC	BLEU*	Time (s)
CC	0.25	0.28	78.10	0.68	0.31	49.20
CR	0.61	0.36	77.03	0.96	0.44	55.73
CS	0.23	0.28	60.10	0.18	0.18	61.83
CSR	0.50	0.20	58.60	0.50	0.30	70.50
FUNPOISON	**0.07**	**0.70**	76.42	**0.06**	**0.56**	70.07

* Since CSR only perturbs comments, we report **BLEU** between the original clean comments and the rewritten poisoned ones. For all other methods, we report **CodeBLEU**, computed between the clean (unpoisoned) code and the rewritten poisoned code.

exceptions, or I/O behavior. These results indicate that FUNPOISON remains semantically harmless while embedding poisoning signals.

Functionality preservation on real-world projects. We further evaluate functionality preservation on a large real-world codebase, Apache Commons Lang (Apache Software Foundation, 2025), which contains 1,908 functions and a comprehensive test suite of 57,764 unit tests. We apply FUNPOISON to a randomly selected 10% of the functions (191/1,908) and evaluate the poisoned project by executing the entire test suite. While the original project passes all tests, the poisoned project also compiles successfully and passes all 57,764 unit tests without failures. This confirms that FUNPOISON preserves functional correctness even when applied to a large real-world codebase.

RQ3: How Robust is FUNPOISON?

Robustness to removal-based methods. We evaluate the robustness of FUNPOISON against four representative poisoning detection and dataset purification methods, with detailed removal rates and post-purification performance reported in Appendix A.3. Among them, KillBadCode and DeCoMa exhibit relatively stronger filtering capabilities: at the 10% poisoning rate, their recalls reach 0.41 and 0.54, reducing the effective poisoning rates after purification to 7.9% and 8.2%, respectively. However, models fine-tuned on the purified datasets still suffer from noticeable performance degradation. Notably, even with only 10% poisoning, the models trained on the purified datasets consistently underperform the base model. These results indicate that FUNPOISON induces persistent poisoning effects that remain effective under state-of-the-art removal-based methods.

Robustness to rewriting-based methods. We evaluate LLM-based rewriting using CodeLlama-7B-Instruct and GPT-4 on 500 poisoned instances (Table 4). While GPT-4 rewrites simpler perturbations more reliably, both models perform poorly against FUNPOISON (CodeLlama: 0.07, GPT-4: 0.06). Rewriting also incurs substantial latency, making web-scale removal costly. These results indicate that the evaluated LLM-based rewriting attacks do not provide a practical removal strategy for FUNPOISON.

Table 5: Detection results under static-analysis and supervised adaptive settings.

Detector	Detection outcome	Accuracy
CodeQL[a]	Same as clean; Rule 32: 4.3%	–
CodeBERT[b]	FPR: 100%	10.39%

[a] CodeQL uses 33 Java rules on 984 instances.
[b] CodeBERT is trained under a 50% poisoned setting with a 100k/32k/32k train/validation/test split.

Robustness to static analysis-based filtering. For static analysis, we evaluate on 984 instances (six-fold duplicated Java HumanEval-X). We compare FUNPOISON only against Clean code, since CoProtector either has negligible effect under partial poisoning or severely degrades compilability, making it trivially detectable. As summarized in Table 5 and detailed in Appendix A.3, CodeQL does not isolate the poisoned samples from clean code under standard rule-based filtering.

Robustness to adaptive detection. We further consider attackers who know FUNPOISON and train a supervised detector to distinguish poisoned samples from clean ones. Under a favorable 50% poisoned setting, the CodeBERT detector fails to obtain a useful accuracy/FPR trade-off and collapses into over-flagging benign code (Table 5). These results do not imply undetectability. Rather, they suggest that supervised detectors struggle to learn stable poison signatures because injected instances are dynamically composed from diverse templates, weak-use statements, and insertion contexts.

Robustness to Code Formatting We evaluate whether automated formatting can neutralize FUNPOISON by applying `clang-format`, whose four-space indentation closely matches the original CodeSearchNet Java style. Formatting only normalizes layout and whitespace while preserving the code; detailed configurations are provided in Appendix A.2. As shown in Fig. 5, formatting fails to remove the poisoning effect: across decoding temperatures and poisoning rates, models fine-tuned on formatted poisoned data consistently underperform both clean fine-tuned and base models, demonstrat-

Table 6: Generalization summary under 10% poisoning. We report representative metrics; complete tables are in Appendix A.4.

Setting	Metric	Base	Clean FT	FUNPOISON
HumanEval-X, CodeLlama-7B	Pass@1, $T = 0.0$	0.29	0.31	0.23
HumanEval-X, CodeLlama-7B-Instruct	Pass@1, $T = 0.0$	0.30	0.38	0.30
MBPP, DeepSeek-1.3B	Pass@1, $T = 0.0$	0.31	0.41	0.16

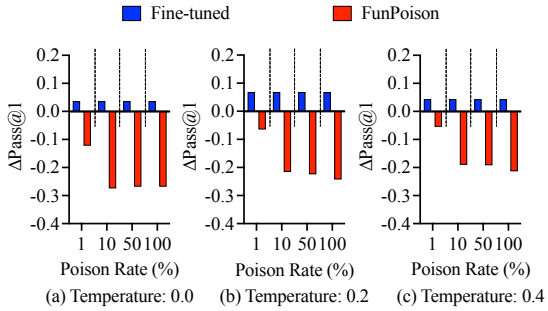

Figure 5: Impact on DeepSeek-Coder-1.3B when full-parameter fine-tuned on datasets that were first poisoned by FUNPOISON and then formatted by CLANG-FORMATTER.

ing robustness to formatter-based normalization.

RQ4: How does the poisoning effect of FUNPOISON vary across settings?

Table 6 summarizes additional executable code-generation settings. FUNPOISON suppresses clean fine-tuning gains in 7B-scale, instruction-tuned, and MBPP experiments. Overall, these experiments focus on executable code-generation tasks, which are directly aligned with the functional behavior targeted by FUNPOISON. Additional sensitivity results for model families, LoRA fine-tuning, insertion count, and template-pool size are reported in Appendix A.7.

6 Related Work

Dataset poisoning and backdoor attacks have been widely studied for classification and language models, but most methods optimize targeted misbehavior or label-space corruption rather than preserving code usability under dataset-level protection. CoProtector (Sun et al., 2022) is the closest prior work for untargeted code dataset poisoning, but its code transformations often break compilability or require full poisoning. In contrast, FUNPOISON treats compilability and behavioral preservation as first-class constraints.

Code dataset protection has also been explored through watermarking and attribution methods (Sun et al., 2023; Xiao et al., 2025; Chen et al., 2026). These techniques help identify dataset misuse after training, whereas FUNPOISON aims to reduce the benefit of unauthorized fine-tuning before or during adaptation. Sanitization methods such as static analysis, formatting, trigger removal, dataset purification, and LLM rewriting are complementary attacker operations; we evaluate them as removal attempts rather than direct baselines because their goal is filtering or rewriting data, not poisoning protected code corpora.

7 Conclusion

We propose a functionality-preserving poisoning framework for protecting code datasets against unauthorized fine-tuning. By injecting execution-inert, compilable fragments into live code paths under strict safety constraints, FUNPOISON embeds training-time signals without affecting program behavior or usability. Experiments and ablations show that FUNPOISON suppresses fine-tuning gains at low poisoning rates because execution-path fragments alter autoregressive supervision, not because templates merely appear in the corpus. Across the evaluated models, code-generation benchmarks, and defenses, functionality preservation and adaptation deterrence can coexist, while the remaining limitations define a clear scope for responsible deployment.

Limitations

Language and task scope

We evaluate FUNPOISON on Java, but this is not a complete cross-language study. Other languages such as C/C++, JavaScript, Go, or Rust require language-specific weak-use pools, parsers, compiler checks, and side-effect filters. Our empirical scope is executable code generation; other code tasks require separate task-specific evaluation.

Insertion-site availability

Compact or highly optimized functions may have few valid execution-safe insertion sites. In CodeSearchNet Java, 80.3% of functions admit valid sites under full coverage, and FUNPOISON operates at the dataset level rather than requiring every

function to be poisoned. Still, site availability may constrain deployment in highly compact codebases.

Removal and training regimes

FUNPOISON is not theoretically unremovable. Our claim is practical: the evaluated removal, rewriting, static-analysis, formatting, and adaptive-detection attempts do not achieve a useful effectiveness, false-positive, semantic-preservation, and cost trade-off. The effect under substantially different training regimes, such as aggressive curriculum learning, large-scale pretraining from scratch, or reinforcement-learning-based adaptation, remains open.

Ethical Considerations

Dataset poisoning is dual-use: the same mechanism that deters non-consensual fine-tuning could disrupt legitimate model training if deployed indiscriminately. We therefore use "defensive" only within the threat model of unauthorized adaptation, not as a universal normative claim.

Responsible deployment should include transparent disclosure in dataset cards, README files, or license addenda; provenance verification through signed manifests or dataset hashes; and controlled clean access for authorized training users. We do not recommend applying FUNPOISON by default to collaborative open-source ecosystems or datasets explicitly intended for unrestricted training. Any research release should include a responsible-use statement and make clear that the method is intended for scoped data-governance settings rather than sabotage of legitimate training.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (U24A20337, 62372228). We used AI assistants only for limited language polishing (e.g., grammar and clarity), minor LaTeX editing, and reference formatting checks. They were not used to generate scientific content, design methods, analyze results, or draw conclusions. All technical contributions, experiments, and interpretations were carried out by the authors.

References

Inc. Amazon Web Services. 2023. CodeWhisperer. site: https://aws.amazon.com/codewhisperer/.

Apache Software Foundation. 2025. Apache commons lang. https://commons.apache.org/proper/commons-lang/. Accessed: 2025-01.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Inc. Beijing Guixin Technology. 2022. aiXcoder. site: https://www.aixcoder.com/.

BigCode Project. 2024. The stack v2: Multilingual, large-scale open-source code corpus. https://huggingface.co/datasets/bigcode/the-stack-v2. Accessed: 2025-09-05.

Bryant Chen, Wilka Carvalho, Nathalie Baracaldo, Heiko Ludwig, Benjamin Edwards, Taesung Lee, Ian M. Molloy, and Biplav Srivastava. 2019. Detecting backdoor attacks on deep neural networks by activation clustering. In *Workshop on Artificial Intelligence Safety 2019 co-located with the Thirty-Third AAAI Conference on Artificial Intelligence 2019 (AAAI-19)*, volume 2301 of *CEUR Workshop Proceedings*, Honolulu, Hawaii. CEUR-WS.org.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yuchen Chen, Yuan Xiao, Chunrong Fang, Zhenyu Chen, and Baowen Xu. 2026. Ducodemark: Dual-purpose code dataset watermarking via style-aware watermark-poison design. *arXiv preprint arXiv:2604.10611*. Accepted to FSE 2026.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547, Online Event. Association for Computational Linguistics.

Saveri Law Firm. 2024. Github copilot intellectual property litigation. Accessed: 2025-09-04.

Free Software Foundation. 2025. Gnu time. https://www.gnu.org/software/time/. Accessed: 2025-09-11.

Inc. GitHub. 2022. GitHub Copilot. site: https://copilot.github.com/.

GitHub Inc. 2025. Codeql: Code analysis engine. https://codeql.github.com/. Accessed: 2025.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi,

Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *arXiv*, abs/2401.14196.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv*, abs/1909.09436.

JaCoCo. 2025. Jacoco java code coverage library. https://www.jacoco.org/jacoco/. Version 0.8.11, Accessed: 2025-09-11.

Legal.io. 2024. Judge throws out majority of claims in github copilot lawsuit. Accessed: 2025-09-04.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023.

LLVM Project. 2023. Clang-format style options. https://clang.llvm.org/docs/ClangFormatStyleOptions.html. Accessed: 2025-01.

InfoQ News. 2022. Class action lawsuit filed against github copilot. Accessed: 2025-09-04.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023. Gpt-4 technical report. Available at https://arxiv.org/abs/2303.08774.

Oracle Corporation. 2025. *javac - The Java Compiler*. Accessed: 2025-09-06.

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

Python Software Foundation. 2025a. difflib — helpers for computing deltas. https://docs.python.org/3/library/difflib.html. Accessed: 2025-09-11.

Python Software Foundation. 2025b. Python programming language, version 3.8.20. https://www.python.org/. Accessed: 2025-09-11.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2023. Code llama: Open foundation models for code. *arXiv*, abs/2308.12950.

Weisong Sun, Yuchen Chen, Mengzhe Yuan, Chunrong Fang, Zhenpeng Chen, Chong Wang, Yang Liu, Baowen Xu, and Zhenyu Chen. 2025. Show me your code! kill code poisoning: A lightweight method based on code naturalness. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2663–2675, Ottawa, Ontario, Canada. IEEE.

Zhensu Sun, Xiaoning Du, Fu Song, and Li Li. 2023. Codemark: Imperceptible watermarking for code datasets against neural code completion models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1561–1572, San Francisco, CA, USA. ACM.

Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. Coprotector: Protect open-source code against unauthorized training usage with data poisoning. In *Proceedings of the ACM Web Conference 2022*, pages 652–660.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, and 49 others. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288. Available at https://arxiv.org/abs/2307.09288.

Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral signatures in backdoor attacks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems*, pages 8011–8021, Montréal, Canada.

Yuan Xiao, Yuchen Chen, Shiqing Ma, Haocheng Huang, Chunrong Fang, Yanwei Chen, Weisong Sun, Yunfeng Zhu, Xiaofang Zhang, and Zhenyu Chen. 2025. Decoma: Detecting and purifying code dataset watermarks through dual channel code abstraction. In *Proceedings of the ACM on Software Engineering*, pages 1701–1724, Trondheim, Norway. ACM.

Yuan Xiao, Yuchen Chen, Jiaming Wang, Wei Song, Jun Sun, Shiqing Ma, Yanzhou Mu, Juan Zhai, Chunrong Fang, Jin Song Dong, and Zhenyu Chen. 2026.

Funpoison. https://github.com/xiaoyuanpigo/FunPoison.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, Long Beach, CA, USA. ACM.

A Appendix

A.1 Additional Method Details

This section gives implementation details for constructing safe and portable poisoned samples. We describe the filtering rules used to reject unsafe fragments (Table 7) and the weak-use expressions used to preserve type-correctness while preventing injected variables from being trivially pruned (Table 8).

A.1.1 Safety Filtering Rules

To ensure that injected fragments do not introduce compilation errors, observable side effects, or semantic deviations, we adopt a conservative filtering policy, summarized in Table 7. The policy consists of two categories of rules.

Conceptual rules, manually curated via semantic reasoning, exclude patterns that may disrupt control flow or introduce hidden dependencies when transplanted across projects, such as explicit control transfers, reflective loading, concurrency primitives, and shared-state mutations.

Programmatic rules are automatically enforced through lightweight static checks and pattern matching, removing fragments that involve I/O operations, non-local state updates, process control, or other observable side effects. This conservative design favors aggressive pruning to guarantee portability and execution safety across diverse codebases.

A.1.2 Weak-Use Expression Pool

To prevent injected fragments from being removed by compilers or preprocessing pipelines due to unused variables or redundant statements, we synthesize weak-use expressions at injection time. These expressions consume declared variables semantically inertly while preserving type correctness.

Weak-use expressions are sampled from a predefined pool of abstract patterns, summarized in Table 8. The pool is organized by variable type and covers common primitive, object, and container categories. All patterns are designed to be side-effect free, excluding I/O, concurrency, and global-state mutations, thereby ensuring portability and compatibility across projects.

A.2 Detailed Experimental Setup

This appendix provides low-level implementation details omitted from the main text to facilitate reproducibility. Unless otherwise specified, all experimental settings are identical across clean and poisoned conditions.

Attack Methods. We evaluate several representative Attack methods against our proposed untargeted poisoning approach, covering detection, static analysis, formatter and rewriting strategies.

Automatic Detection/Removal. We evaluate FUNPOISON against representative detection and purification methods. **Spectral Signature (SS)** (Tran et al., 2018) detects poisoned samples by analyzing anomalous spectral components in latent representations via singular value decomposition. **Activation Clustering (AC)** (Chen et al., 2019) clusters activations using k-means and flags small outlier clusters as poisoned. **KillBadCode** (Sun et al., 2025), the SOTA for code, identifies trigger tokens by measuring perplexity changes under an n-gram language model and removes all samples containing them. Finally, **DeCoMa** (Xiao et al., 2025) targets code dataset watermark detection by abstracting code into dual-channel templates and eliminating anomalous trigger–target pairs via frequency-based outlier analysis.

Static Analysis. We adopt CodeQL (GitHub Inc., 2025), a widely used static analysis framework, to examine poisoned code. Specifically, we apply 33 curated Java queries across five categories (Table 9): (i) *redundancy/dead code*, covering unused parameters, redundant null checks, and unreachable branches; (ii) *API misuse/correctness*, targeting incorrect method signatures and contract violations; (iii) *generics/readability*, detecting type mismatches and style issues; (iv) *concurrency*, identifying unsafe synchronization and locking patterns; and (v) *exceptions/control flow*, capturing unreachable catch blocks and unhandled exceptions. Notably, we incorporate *all* CodeQL rules related to redundancy and dead code, ensuring full coverage of potential signals. In our evaluation, CodeQL serves two roles: (1) validating that injected fragments preserve compilability and coding conventions, and (2) testing whether static analysis can effectively detect and prune poisoning artifacts.

Formatter-based Normalization. To evaluate robustness against formatting-based sanitization, we apply automated code normalization using `clang-format`. We note that an attacker may employ arbitrary formatters; our goal is therefore not to assume a specific attacker choice, but to test whether poisoning signals persist under representative, structure-preserving formatting. We choose

Table 7: Consolidated unsafe code patterns filtered during template preprocessing. We distinguish between *conceptual* rules (manually curated for semantic safety) and *programmatic* rules (automatically enforced by regex/static checks).

Category	Unsafe Pattern (motivation → examples)
Conceptual (manually defined safety rules)	
Package/API	Risky dependencies → `java.io`, `java.net`, `java.nio.file`, `java.lang.reflect`, `sun.misc.Unsafe`
Dangerous Calls	Termination or reflective loading → `System.exit()`, `Runtime.getRuntime().exec()`, `Runtime.getRuntime().halt()`, `Class.forName()`
Concurrency	Uncontrolled synchronization → `Thread.notify()`, `Thread.notifyAll()`, `Thread.wait()`
Optional Misuse	Risk of NPE → `reduce(...).get()`, `findFirst().get()`, `findAny().get()`, bare `Optional.get()`
Control Flow	Premature termination → explicit `return`, `throw`, `break`, `continue`
Assertion	Non-portable checks → use of `assert` statements
Data Mutation	Side-effectful updates → container `add`, `put`, `remove`, `clear`
String/Builder	Hidden allocation → unsafe concatenation with `StringBuilder+toString()`
Unsafe Equality	Semantic mismatch → `"abc".equals(obj)` where `obj` is not a string
Null Checks	Redundancy → `x = new T(); if (x == null) ...`
Top-level Types	Noise → empty/unused `class`/`interface`/`enum` declarations
Reserved Names	Shadowing JDK core types → `Object`, `List`, `Map`, etc.
Programmatic (regex/static enforced rules)	
OS / Process	Unsafe process creation → `new ProcessBuilder(...).start()`
File / Network I/O	Direct I/O and sockets → `new FileReader(...)`, `new Socket(...)`, `URLConnection`, `Files.write/delete/move/copy(...)`
Printing / Logging	Observable side effects → `System.out.println/printf`, `System.err.println/printf`
Threading / Sync APIs	Heavy concurrency primitives → `Executor`, `ExecutorService`, `Semaphore`, `Lock`
Non-local Writes	State mutation outside local scope → field/array writes (`x.f = ...`, `x[i] = ...`) where x non-local
Non-local Mutators	Side-effectful non-local calls → `set*`/`put*`/`add*`/`remove*`/`update*`
Stub Conflicts	Name collision → same-file stubs shadowing core JDK types

Table 8: Type-specific weak-use expression categories. Each abstract category consolidates multiple concrete patterns (all enumerated), ensuring complete coverage while avoiding redundancy.

Variable Kind	Weak-use Categories → Concrete Patterns
Boolean	Logical boxing → `Boolean.valueOf(v)`, `System.identityHashCode(Boolean.valueOf(v))` Trivial logical eval → `v && true`, `v || false`, `!(v)==false`, `Boolean.valueOf(true)`
Char	Numeric promotion → `Integer.valueOf((int)v)`, `System.identityHashCode(Character.valueOf(v))`
Numeric (int / short / byte)	Arithmetic identity → `Math.abs(v)`, `Math.max(v,v)`, `Math.min(v,v)`, `v+0` Bitwise safe ops → `((int)v)|0`, `((int)v)&-1` Boxing / identity → `System.identityHashCode(Integer.valueOf((int)v))`
Numeric (long)	Arithmetic identity → `Math.abs(v)`, `Math.max(v,v)`, `Math.min(v,v)`, `v+0L` Bitwise safe ops → `(v|0L)`, `(v&-1L)` Boxing / identity → `System.identityHashCode(Long.valueOf(v))`
Numeric (float / double)	Arithmetic identity → `Math.abs(v)`, `Math.max(v,v)`, `Math.min(v,v)`, `v+0` Representation access → `Math.nextAfter(v,v)`, `Double.doubleToRawLongBits(v)` Boxing / identity → `System.identityHashCode(Double.valueOf(v))`
String / CharSequence	Structural query → `String.valueOf(v).length()`, `(int)String.valueOf(v).chars().count()` Emptiness → `String.valueOf(v).isEmpty()` Identity → `System.identityHashCode(String.valueOf(v))`
Optional-like wrappers	Existence check → `(int)Stream.ofNullable(v).count()`, `Stream.ofNullable(v).findAny().isPresent()`
Array	Structural hashing → `Arrays.hashCode(v)`, `Arrays.deepHashCode(new Object[]v)` String identity → `System.identityHashCode(Arrays.deepToString(new Object[]v))`
Collection-like	Guarded structural queries → `{o=v; if(o instanceof Collection) ((Collection)o).size(); else 0;}`, `{o=v; if(o instanceof Collection) ((Collection)o).isEmpty(); else true;}`
Map-like	Guarded structural queries → `{o=v; if(o instanceof Map) ((Map)o).size(); else 0;}`, `{o=v; if(o instanceof Map) ((Map)o).keySet().size(); else 0;}`, `{o=v; if(o instanceof Map) ((Map)o).values().size(); else 0;}`
Generic Object	Identity / hash → `System.identityHashCode(v)`, `Objects.hashCode(v)`, `Integer.valueOf(Objects.hashCode(v))`

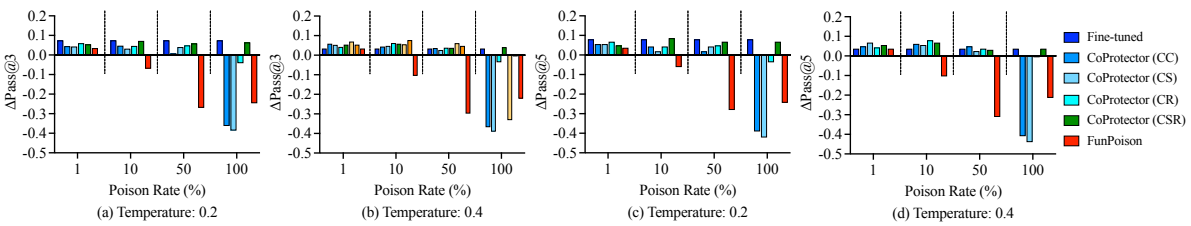

Figure 6: Poisoning effects (ΔPass@3 and ΔPass@5) of full-parameter fine-tuning DeepSeek-Coder-1.3B on datasets poisoned by FUNPOISON and baseline methods.

Table 9: Categorized Java CodeQL rules relevant to redundancy, correctness, concurrency, and control flow.

Category and Rule IDs (No.)
Redundancy / Dead Code: (1) unused-parameter, (2) unused-label, (3) unused-format-argument, (4) unused-container, (5) unused-reference-type, (6) useless-null-check, (7) useless-tostring-call, (8) useless-type-test
API Misuse / Correctness: (9) equals-on-unrelated-types, (10) unchecked-cast-in-equals, (11) tostring-typo, (12) whitespace-contradicts-precedence, (13) wrong-equals-signature, (14) wrong-comparator-signature, (15) wrong-object-serialization-signature, (16) wrong-readresolve-signature, (17) wrong-junit-suite-signature, (18) wrong-swing-event-adapter-signature
Generics & Readability: (19) type-variable-hides-type, (20) type-bound-extends-final, (21) type-mismatch-access, (22) type-mismatch-modification, (23) underscore-identifier, (24) unknown-javadoc-parameter
Concurrency: (25) unreleased-lock, (26) unsafe-sync-on-field, (27) unsynchronized-getter, (28) wait-on-condition-interface, (29) unsafe-double-checked-locking, (30) unsafe-double-checked-locking-init-order
Exceptions / Control Flow: (31) unreachable-catch-clause, (32) uncaught-number-format-exception, (33) unsafe-get-resource

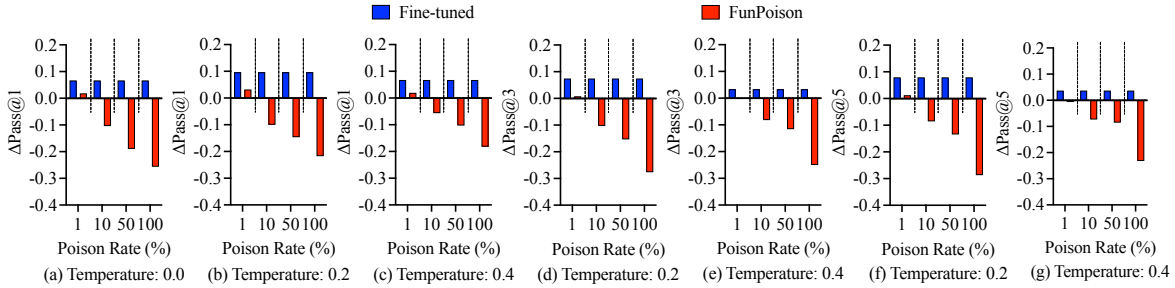

Figure 7: Impact on DeepSeek-Coder-1.3B when full-parameter fine-tuned on datasets that were first poisoned by FUNPOISON and then purified by DeCoMa.

`clang-format` because it is widely used and can be configured to closely match the original formatting style of CodeSearchNet (predominantly four-space indentation), avoiding confounding distribution shifts introduced by incompatible formatting conventions (e.g., two-space indentation).

We use a customized Java formatting configuration based on the Google style, with `IndentWidth=4`, `ContinuationIndentWidth=4`, `ColumnLimit=0`, attached braces, and disabled short-statement collapsing, ensuring that only layout and whitespace are modified while program structure are preserved. Formatting is applied to the entire dataset using a parallelized pipeline with per-file timeouts; files that fail formatting are kept unchanged. All samples are successfully formatted in our experiments

LLM-based Rewriting. We investigate whether large language models can be leveraged to rewrite poisoned code while retaining its original functionality. To this end, we examine two representative families: *Code Llama* (Rozière et al., 2023), an open-source extension of Llama 2 (Touvron et al., 2023) specialized for programming tasks, and *GPT-4* (OpenAI, 2023), a proprietary model from OpenAI noted for its advanced reasoning and code synthesis capabilities. Our experiments employ the 7B-parameter Code Llama-Instruct variant and assess both models' ability to detect and eliminate injected poisoning patterns without compromising code correctness.

Training Settings. Following previous studies (Li et al., 2023; Guo et al., 2024), we adopt the following settings for model fine-tuning: 2 training epochs, a maximum sequence length of 1024, a learning rate of 2×10^{-5}, a per-device batch size of 8, weight decay of 0.1, and a cosine learning rate scheduling strategy. All experiments are conducted on a server equipped with two NVIDIA RTX 3090 GPUs (24 GB each) and a 48-core Intel Xeon Silver 4310 CPU with 125 GB of RAM.

A.2.1 Evaluation Metrics

Poison Effect. To assess poisoning effectiveness, we focus on Pass@k (Chen et al., 2021), which has become the de facto standard for evaluating functional correctness of code generation models. Pass@k reports the probability that at least one of the top-k generated candidates passes all unit tests. We use $k = 1, 3, 5$ to capture both strict correctness (Pass@1) and more permissive scenarios (Pass@3, Pass@5). All experiments are conducted on HumanEval-X (Zheng et al., 2023), a multilingual benchmark derived from HumanEval (Chen et al., 2021) that provides unit-test–based evaluation across programming languages. Functional correctness is the primary indicator of whether poisoning successfully impairs model utility, making

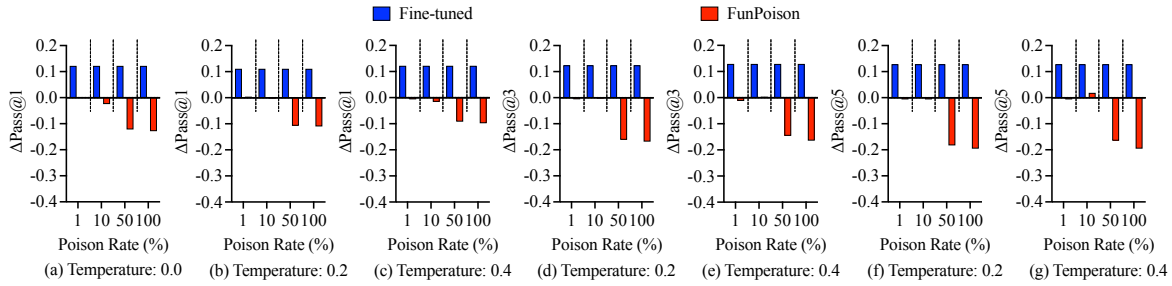

Figure 8: Poisoning effects of full-parameter fine-tuning StarCoder-1B on datasets poisoned by FUNPOISON.

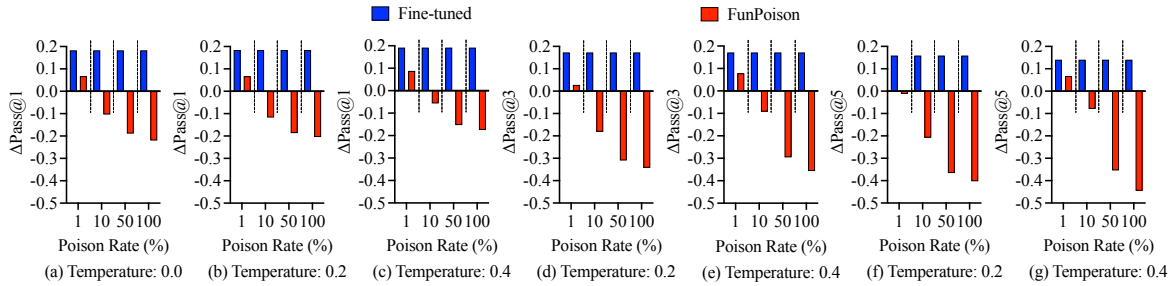

Figure 9: Poisoning effects of LoRA fine-tuning DeepSeek-Coder-6.7B on datasets poisoned by FUNPOISON.

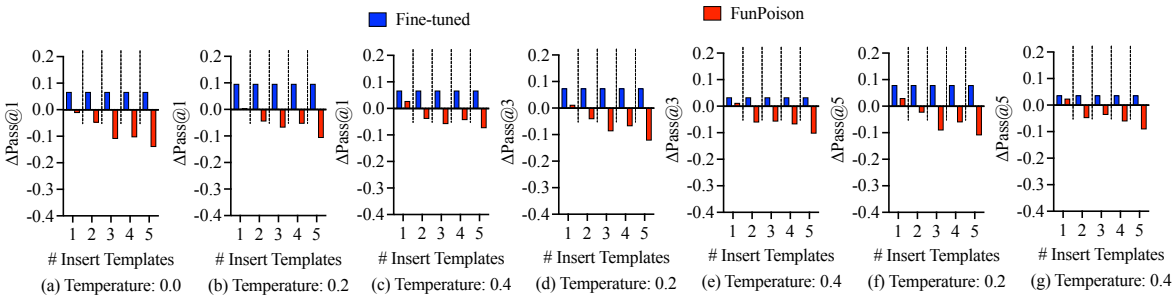

Figure 10: Poisoning effect of FUNPOISON on DeepSeek-Coder-1.3B at a 10% injection ratio, evaluated across varying numbers of insertion templates.

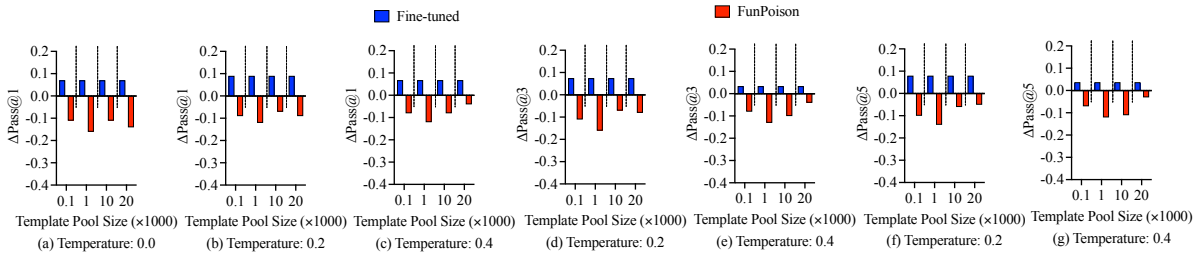

Figure 11: Poisoning effect of FUNPOISON on DeepSeek-Coder-1.3B at a 10% injection ratio, evaluated across varying sizes of the templates pool.

Pass@k a particularly critical metric in our study.

Code Functionality and Quality. To quantify the similarity between poisoned and reference code, we adopt two complementary metrics. **BLEU** (Papineni et al., 2002) measures n-gram overlap between generated and reference code, reflecting surface-level similarity. **CodeBLEU** (Ren et al., 2020) extends BLEU by further incorporating syntax- and semantics-aware components (e.g., weighted syntax and data-flow matches), offering a more comprehensive evaluation of code quality. We report both BLEU and CodeBLEU on 1,000 randomly sampled functions from the CSN test set.

To evaluate whether injected fragments remain semantically harmless while preserving runtime stability, we perform dynamic analysis along four dimensions. First, **performance overhead** quantifies runtime cost in terms of latency and memory (e.g., p95 latency and peak memory consumption). Second, **coverage analysis** monitors line-level execution to detect any deviations in control flow or test coverage. Third, **stability analysis** measures execution variability using jitter ratios (p95–p50)/p50 and regression rates, indicating whether injected code destabilizes runtime behavior. Finally, **behavior consistency** compares outputs between clean and poisoned variants—including stdout, stderr, exceptions, and file interactions, to ensure functional equivalence. Together, these metrics provide a holistic view of efficiency, correctness, and reliability, verifying that poisoning preserves functionality while embedding persistent signals. We conduct dynamic analysis using GNU time (Free Software Foundation, 2025) for measuring performance and stability, JaCoCo (JaCoCo, 2025) for coverage instrumentation, Python 3.8.20 with the `difflib` library (Python Software Foundation, 2025b,a) for behavior consistency, and the Java compiler (javac) and JVM (Oracle Corporation, 2025) with fixed memory configurations to ensure reproducibility.

A.3 Robustness Details

This section provides the detailed robustness results that support the main-text discussion. We first report removal-attack outcomes and static-analysis results in Tables 10 and 11, and then provide the full fine-tuning results after applying KillBadCode and DeCoMa purification in Figures 12 and 13.

Table 10: False positive rate (FPR), recall, and effective poisoning rate after purification (A. Poi. Rate) for FUNPOISON under different removal attacks and poisoning rates.

Attack	Metric	Poisoning Rate (%)			
		1	10	50	100
SS	FPR	0.07	0.06	0.06	0.13
	Recall	0.06	0.17	0.09	0.06
	A. Poi. Rate	1.0	8.9	49.2	81.5
AC	FPR	0.14	0.42	0.65	0.50
	Recall	0.13	0.72	0.15	0.29
	A. Poi. Rate	1.0	5.1	70.8	85.3
KillBadCode	FPR	0.24	0.24	0.21	0.13
	Recall	0.38	0.41	0.40	0.39
	A. Poi. Rate	0.8	7.9	43.2	74.1
DeCoMa	FPR	0.42	0.43	0.44	0.35
	Recall	0.49	0.54	0.83	0.83
	A. Poi. Rate	0.9	8.2	23.3	51.6

* Although the poisoning configuration is set to 100% of the dataset, the effective coverage is constrained by the availability of valid insertion sites. As a result, the maximum effective poisoning rate reaches 80.3%.

Table 11: Static analysis results.

Rule ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Clean	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FUNPOISON	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rule ID	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
Clean	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4.3	0
FUNPOISON	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4.3	0

A.4 Extended Mechanism and Generalization Results

A.5 Adaptive Detector Details

For static analysis, CodeQL applies 33 Java rules to the 984-instance evaluation set. Its alerts are identical for clean and poisoned code: both trigger only Rule 32 at 4.3%, consistent with Table 11. This indicates that the standard static-analysis rule set does not isolate poisoned samples from benign code.

For the supervised detector, we fine-tune a CodeBERT classifier on a 50% poisoned setting with 100k training, 32k validation, and 32k test examples. The detector fails to find a useful operating point: thresholds that increase recall also over-flag benign code, producing the 100% false-positive regime reported in Table 5. We therefore interpret this result as evidence that the classifier does not learn a stable and generalizable poison signature from the dynamic combinations of templates, weak-use statements, and insertion contexts, not as evidence of theoretical undetectability.

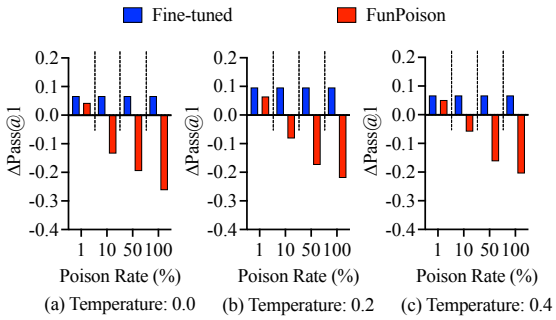

Figure 12: Impact on DeepSeek-Coder-1.3B when fine-tuned on datasets first poisoned by FUNPOISON and then purified by KillBadCode.

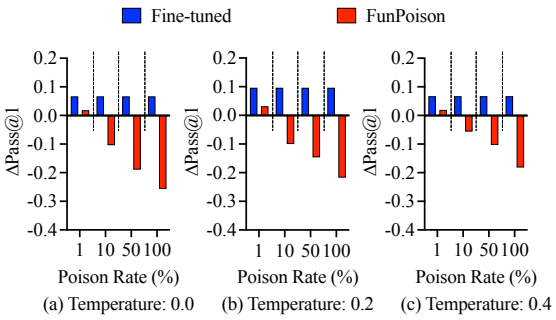

Figure 13: Impact on DeepSeek-Coder-1.3B when fine-tuned on datasets first poisoned by FUNPOISON and then purified by DeCoMa.

Table 12: Structural pattern occurrence and failure correlation across poisoning rates. Values are reported as fail/total occurrences over generated samples.

Rate	Temperature	Weak-use fail/total	Template fail/total
1%	0.0	125 / 170	385 / 520
	0.2	126 / 179	389 / 537
	0.4	140 / 183	470 / 604
10%	0.0	415 / 435	755 / 785
	0.2	443 / 468	735 / 781
	0.4	437 / 460	741 / 778
50%	0.0	490 / 505	760 / 795
	0.2	487 / 507	754 / 792
	0.4	461 / 478	748 / 784
100%	0.0	735 / 760	780 / 815
	0.2	740 / 761	785 / 809
	0.4	729 / 747	791 / 810
Total	–	5328 / 5653	8093 / 8810

A.6 Pass 3 and 5 Results for RQ1 (How effective is FUNPOISON at degrading model performance on code generation?)

As shown in Fig. 6, FUNPOISON exhibits stable poisoning effects across decoding temperatures (0.0/0.2/0.4): when the poisoning rate reaches 10%, ΔPass@3/ΔPass@5 drop clearly below zero, indicating that models fine-tuned on poisoned data perform substantially worse than the baseline; as the poisoning rate increases to 50% and 100%, the degradation further intensifies. In contrast, when the poisoning rate is below 100%, other baselines generally have a weak impact on performance. In some settings, models fine-tuned on these baselines even outperform the clean fine-tuned model.

A.7 RQ4: How does the poisoning effect of FUNPOISON vary across settings?

Effectiveness on other code LLMs. We also evaluate the poisoning effectiveness of FUNPOISON on other Code LLMs, such as StarCoder. Figure 8 presents the experimental results on StarCoderBase-1b. As shown, FUNPOISON successfully achieves poisoning across decoding temperatures of 0.0, 0.2, and 0.4. Remarkably, even with only a 1% poisoning rate, the effectiveness of fine-tuned StarCoderBase degrades to a level comparable to that of the base model.

Effectiveness under other fine-tuning method. In addition, considering that adversaries may exploit alternative training methods (e.g., efficient fine-tuning) to adapt models using the protected dataset, we further evaluate the poisoning effectiveness of FUNPOISON under LoRA fine-tuning of DeepSeek-6.7B, with the results shown in Figure 9. Across decoding temperatures of 0.0, 0.2, and 0.4, FUNPOISON consistently maintains poisoning effectiveness: the pass rate of the fine-tuned model with FUNPOISON drops significantly below that of normal fine-tuning and approaches the performance of the base model.

Sensitivity to the number of injected templates. With the injection ratio fixed at 10%, we further examine the impact of template diversity by varying the number of injected templates from 1 to 5. As shown in Figure 10, increasing the number of templates generally strengthens the poisoning effect of FUNPOISON, but the gain saturates once

Table 13: HumanEval-X results for 7B and instruction-tuned models.

Model	Variant	Temperature	Pass@1	Pass@3	Pass@5
CodeLlama-7B	Base	0.0	0.29	0.29	0.29
	FUNPOISON (10%)	0.0	0.23	0.23	0.23
	Clean fine-tuned	0.0	0.31	0.31	0.31
	Base	0.2	0.29	0.43	0.49
	FUNPOISON (10%)	0.2	0.22	0.36	0.42
	Clean fine-tuned	0.2	0.32	0.39	0.41
	Base	0.4	0.23	0.44	0.53
	FUNPOISON (10%)	0.4	0.23	0.40	0.48
	Clean fine-tuned	0.4	0.31	0.45	0.51
CodeLlama-7B-Instruct	Base	0.0	0.30	0.30	0.30
	FUNPOISON (10%)	0.0	0.30	0.30	0.30
	Clean fine-tuned	0.0	0.38	0.38	0.38
	Base	0.2	0.29	0.39	0.44
	FUNPOISON (10%)	0.2	0.30	0.40	0.43
	Clean fine-tuned	0.2	0.38	0.47	0.50
	Base	0.4	0.23	0.40	0.46
	FUNPOISON (10%)	0.4	0.27	0.40	0.45
	Clean fine-tuned	0.4	0.32	0.48	0.54

Table 14: Additional code-generation benchmark results for DeepSeek-Coder-1.3B under 10% poisoning.

Benchmark	Variant	Temperature	Pass@1	Pass@3	Pass@5
MBPP	Base	0.0	0.31	0.47	0.52
	Clean fine-tuned	0.0	0.41	0.41	0.41
	FUNPOISON	0.0	0.16	0.16	0.16
	Base	0.2	0.35	0.45	0.48
	Clean fine-tuned	0.2	0.40	0.47	0.50
	FUNPOISON	0.2	0.18	0.29	0.35

the number reaches three. This suggests that using three templates provides a reasonable trade-off: it introduces sufficient diversity to achieve strong poisoning while avoiding the overhead of inserting excessive variants.

Sensitivity to the template pool size. We study how the size of the template pool affects the poisoning strength of FUNPOISON, as shown in Fig 11. Specifically, we vary the pool size $K \in \{100, 1{,}000, 10{,}000, 20{,}000\}$ while keeping other settings fixed. Across all pool sizes, FUNPOISON consistently produces strong poisoning effects. Even the smallest pool ($K = 100$) substantially reduces Pass@1 at $T = 0.0$ from the clean fine-tuned baseline of 0.38 to 0.20. Increasing the pool size yields comparable or slightly stronger degradation (e.g., Pass@1 of 0.15 at $K = 1{,}000$ and 0.20 at $K \approx 8{,}000$), with similar trends observed for Pass@3 and Pass@5 across decoding temperatures. Overall, the differences across pool sizes are small relative to the significant degradation introduced by FUNPOISON itself, indicating that template-pool size has only a minor impact on poisoning strength. These results suggest that FUNPOISON remains highly effective even with modest template diversity, and that increasing K beyond a few hundred offers only marginal additional benefit.

A.8 Case Studies

Figures 14 and 15 illustrate representative code examples poisoned by FUNPOISON under the default setting, where each function is injected with up to 0–3 templates. As shown, FUNPOISON introduces only minimal and localized modifications: the original control flow and logic remain intact, while injected fragments appear as benign auxiliary code (e.g., object instantiation, identity queries) that naturally blend into real-world implementations. These examples highlight that FUNPOISON preserves compilability and runtime behavior with low visual and structural footprint.

```
protected void setOffsetAndLength
    (long offset, int length) throws IOException {
    this.offset = offset;
    this.length = length;
    this.position = 0;

    if (subStream.position() != offset) {
        subStream.seek(offset);
    }
}
```

(a) Original Code

```
class ExampleUtils {
    public void setup() {}
}
protected void setOffsetAndLength
    (long offset, int length) throws IOException {
    this.offset = offset;
    this.length = length;
    this.position = 0;

    if (subStream.position() != offset) {
        subStream.seek(offset);
        ExampleUtils exampleUtils = new ExampleUtils();
        exampleUtils.setup();
        java.lang.System.identityHashCode(exampleUtils);
    }
}
```

(b) Code Poisoned by FunPoison

Figure 14: Example of FUNPOISON

```
public Set<ConstraintViolation> validate() {
    Set<ConstraintViolation> errors =
        new LinkedHashSet<ConstraintViolation>();
    for (int record = 1; record <= 3; ++record) {
        errors.addAll(validate(record));
    }
    return errors;
}
```

(a) Original Code

```
class Fos {
    public void close() {}
}
public Set<ConstraintViolation> validate() {
    Set<ConstraintViolation> errors =
        new LinkedHashSet<ConstraintViolation>();
    Fos fos = new Fos();
    fos.close();
    java.lang.System.identityHashCode(fos);
    for (int record = 1; record <= 3; ++record) {
        errors.addAll(validate(record));
    }
    return errors;
}
```

(b) Code Poisoned by FunPoison

Figure 15: Example of FUNPOISON