A Small Leak Sinks All: Exploring the Transferable Vulnerability of Source Code Models

Weiye Li[a], Wenyi Tang[a,1]

[a]*Sichuan University, Chengdu, China*

Abstract

Source Code Model (SCM) aims to learn the proper embeddings from source codes, demonstrating significant success in various software engineering or security tasks. The recent explosive development of Large Language Model (LLM) extends the family of SCMs, bringing LLMs for code (LLM4Code) that revolutionize development workflows. Investigating different kinds of SCM vulnerability is the cornerstone for the security and trustworthiness of AI-powered software ecosystems, however, the fundamental one, transferable vulnerability, remains critically underexplored. Existing studies neither offer practical ways, *i.e.* require access to the downstream classifier of SCMs, to produce effective adversarial samples for adversarial defense, nor give heed to the widely used LLM4Code in modern software development platforms and cloud-based integrated development environments. Therefore, this work systematically studies the intrinsic vulnerability transferability of both traditional SCMs and LLM4Code, and proposes a victim-agnostic approach to generate practical adversarial samples. We design a Hierarchical Adaptive Bandit-based Intelligent method for Transferable Attack (HABITAT), consisting of a tailored perturbation-inserting mechanism and a hierarchical Reinforcement Learning (RL) framework that adaptively selects optimal perturbations without requiring any access to the downstream classifier of SCMs. Furthermore, an intrinsic transferability analysis of SCM vulnerabilities is conducted, revealing the potential vulnerability correlation between traditional SCMs and LLM4Code, together with fundamental factors that govern the success rate of victim-agnostic transfer attacks. These findings of SCM vulnerabilities underscore the critical focal points for developing robust defenses in the future. Experimental evaluation demonstrates that our constructed adversarial examples crafted based on traditional SCMs achieve up to 64% success rates against LLM4Code, representing over 15% improvement over the existing state-of-the-art method.

Keywords:
Software Ecosystem Security, Source code model, LLMs for code, Transferable vulnerability

1. Introduction

The advent of SCMs Feng et al. (2020); Wang et al. (2021); Guo et al. (2022, 2020) has introduced powerful capabilities of software engineering and security research. These models, inspired by breakthroughs in natural language processing like BERT Devlin et al. (2019), aim to encode source codes into proper embeddings that capture their semantic and syntactic properties. In task-specific software applications, a SCM outputs its encoded embeddings to downstream classifiers, such as code classification Zhang et al. (2019); Alon et al. (2019), code generation Chen et al. (2021); Wang et al. (2021), code repair Chen et al. (2019); Lutellier et al. (2020), clone detection Svajlenko et al. (2014), vulnerability detection Zhou et al. (2019), etc. Traditional SCMs primarily focused on encoder-only and encoder-decoder architectures, demonstrating effectiveness in understanding and manipulating code for specific downstream tasks Nijkamp et al. (2022). The emergence of LLMs has catalyzed a paradigm shift toward decoder-only architectures trained on large-scale text and code repositories such as GitHub Chen et al. (2021), expanding the scope from task-specific code understanding to general-purpose code generation and reasoning. This evolution has given rise to LLM4Code systems Roziere et al. (2023); Lozhkov et al. (2024); Guo et al. (2024); Li et al. (2023), which leverage massive pre-training to achieve unified capabilities across diverse coding scenarios Wan et al. (2024). The transition has dramatically expanded the scope and sophistication of code understanding and generation capabilities. Yet despite these remarkable advances, the security implications of this architectural evolution remain critically underexplored.

One particular concern is the SCM's vulnerability to adversarial examples, for instance, minor syntactic changes (e.g., adding dead code, renaming variables) can cause a misclassification, thus exposing systemic risks in downstream software engineering applications. The major body of current works has paid attention to the basic adversarial examples of SCM, but overlooked the transferable ones Papernot et al. (2016); Liu et al. (2016). Considering the transfer attack scenario, a source SCM could be used to generate adversarial examples against a victim SCM and mislead the victim SCM's downstream classifier. Recent research Yang et al. (2024) has discovered that such transferable vulnerabilities exist among traditional SCMs, where adversarial examples crafted on one source SCM can transfer across multiple downstream classifiers. Based on their

*Corresponding author.
URL: wyli@stu.scu.edu.cn (Weiye Li), wtang@scu.edu.cn (Wenyi Tang)

reliance on victim classifier information, these transfer attacks fall into two categories: victim-aware approaches that leverage downstream model information, and victim-agnostic methods that operate without victim knowledge. This means that despite architectural differences across SCM systems, they share fundamental vulnerabilities that enable systematic compromise of the entire ecosystem, suggesting that structural diversity provides insufficient protection against powerful transfer attacks. In detail, similar to the above example, a maliciously crafted input that triggers incorrect code classification in one SCM could systematically evade downstream security classifiers across different platforms, including the critical transition from traditional encoder-only models to modern LLM4Code systems deployed in production, creating cascading security failures across entire software ecosystems Zou et al. (2023). The investigation of these transferable examples helps us better understand the fundamental limitations of current models and develop more robust training procedures across the entire SCM ecosystem.

Current adversarial attack methods against SCMs Nguyen et al. (2023); Yang et al. (2022); Zhang et al. (2020) predominantly follow victim-aware paradigms that require prior knowledge of target architectures and model parameters. While these approaches can achieve high attack success rates on specific systems, their practical implementation is severely constrained by the difficulty of obtaining detailed victim information in real-world scenarios. This reliance on victim-specific knowledge makes these attacks hard to deploy systematically across the landscape of SCM systems, from traditional encoder-only models to modern LLM4Code architectures. Consequently, the limited transferability of existing victim-aware methods has failed to provide effective adversarial examples for developing robust defense strategies. Without practical attack vectors that can realistically threaten SCM systems, security researchers lack the necessary tools to evaluate and strengthen defenses against systematic vulnerabilities.

As SCMs increasingly integrate into critical development workflows, understanding and mitigating these transferable vulnerabilities has become essential for maintaining software security. However, to our best knowledge, there is no study into the fundamental mechanisms governing vulnerability transferability between traditional SCMs and LLM4Code systems. The security community lacks comprehensive empirical frameworks that can predict and explain why certain adversarial perturbations successfully transfer across different architectural paradigms. Recent evidence suggests that vulnerability patterns exhibit unexpected cross-system transferability, yet the underlying factors driving this phenomenon remain poorly understood. This knowledge gap hampers the development of robust defenses and leaves critical vulnerabilities in production deployments inadequately characterized, highlighting the urgent need for systematic investigation into the empirical foundations of cross-system vulnerability transfer.

To bridge these gaps, we propose a victim-agnostic approach HABITAT for defense that achieves high transferability while assuming absolute zero access to downstream models and tasks. We present a novel hierarchical reinforcement learning framework that systematically generates powerful and practical adversarial examples through a multi-stage process guided by source SCMs encoders. Our approach begins by reformulating adversarial code modification as a hierarchical contextual bandit problem, where we strategically select insertion positions based on semantic salience at the global level, while choosing from six types of syntax-preserving transformations at the local level through position-specific bandits. To enhance cross-model generalization, we construct a transferable attack memory that records not only successful transformation strategies but also their contextual reward distributions from previous attacks. This memory serves as a foundation for our multi-model guidance mechanism, which employs a model-selection bandit to dynamically determine the most applicable attack strategies from previous models' experiences for each position in the current victim SCM, resulting in per-position attack plans tailored to the victim SCM's latent vulnerability surface. We then evaluate these adversarial examples which crafted solely through access to traditional SCMs, against LLM4Code while maintaining semantic equivalence. Our comprehensive evaluation spans multiple traditional SCMs, LLM4Code systems, and code classification tasks, achieving up to 64% success rates against LLM4Code and over 15% improvement over exisiting state-of-the-art on traditional SCMs, demonstrating that our method effectively bridges the existing transferability gap between traditional SCMs and modern LLM4Code systems. Meanwhile, we introduced several dominant factors that may caused this transferability, including the insert positions, the different insertion types success rates, local via global attentions distribution, the code snippet's complexity, and its semantic comprehension disparities.

Therefore, our contributions can be summarized as follows:

- To the best of our knowledge, this is the first systematic study about the transferable vulnerabilities of both traditional SCMs and LLM4Code.
- We proposed the HABITAT method, which combines a tailored perturbation-insertion mechanism with a hierarchical RL framework that adaptively selects optimal perturbations without requiring access to downstream classifiers of SCMs.
- We conducted an intrinsic transferability analysis of SCM vulnerabilities, revealing potential vulnerability correlations between traditional SCMs and LLM4Code, along with dominant factors governing the success rate of victim-agnostic transfer attacks.
- We evaluate the effectiveness of our proposed method, showing that our constructed adversarial samples transferred from traditional SCMs to the LLM4Code achieve up to 64% success rate, and outperform the state-of-the-art methods on different evaluation metrics.

2. Related Work

Traditional SCMs: Traditional SCMs evolved from encoder-only and encoder-decoder architectures designed for

code understanding. CodeBERT Feng et al. (2020) pioneered the adaptation of bidirectional encoders to code, while GraphCodeBERT Guo et al. (2020) incorporated data flow information for enhanced semantic comprehension. UnixCoder Guo et al. (2022) unified multiple code tasks in a single pre-trained model, and CodeT5 Wang et al. (2021) implemented a transformer encoder-decoder architecture for both generation and understanding tasks. These models demonstrated effectiveness across applications including code summarization Hu et al. (2018), search Gu et al. (2018), and defect detection Zhou et al. (2019), while Code2vec Alon et al. (2019) and ASTNN Zhang et al. (2019) innovated with structural code representations.

Evolution of LLM4Code: LLM4Code marked a shift from task-specific encoder architectures to generative decoder-only models with larger parameter and training data. Codex Chen et al. (2021) demonstrated that large-scale pre-training on GitHub repositories enabled code synthesis from natural language. This evolution continued with CodeGen Nijkamp et al. (2022), which investigated scaling properties of autoregressive models, while open-source contributions included StarCoder Li et al. (2023); Lozhkov et al. (2024), trained on over 80 programming languages. DeepSeek-Coder Guo et al. (2024) advanced the field by combining extensive pre-training with instruction tuning, and CodeLlama Roziere et al. (2023) adapted the Llama 2 Touvron et al. (2023) architecture specifically for programming tasks, achieving state-of-the-art performance across numerous benchmarks Wan et al. (2024).

Adversarial Attacks on Code Models: As SCMs entered development workflows, adversarial vulnerabilities became critical concerns. Zhang et al. Zhang et al. (2020) pioneered gradient-based adversarial examples, while Yang et al. Yang et al. (2022) developed natural adversarial examples preserving functionality. ALERT demonstrated variable renaming significantly impacts model predictions. Nguyen et al. Nguyen et al. (2023) explored code completion vulnerabilities, and Ramakrishnan et al. Ramakrishnan and Albarghouthi (2022) investigated backdoor attacks. These span from white-box methods requiring complete model access to victim-agnostic techniques operating without target knowledge.

Transfer Attacks: Transfer attacks allow crafting examples on accessible models that transfer to inaccessible targets. Originally studied in computer vision Papernot et al. (2016); Liu et al. (2016), code model transferability faces unique challenges from programming languages' structured nature. Yang et al. Yang et al. (2024) discovered transferable vulnerabilities across different SCM architectures, indicating shared weaknesses. Yefet et al. Yefet et al. (2020) identified cross-model transferable perturbations for code naming tasks. However, factors influencing transferability between traditional SCMs and LLM4Code remain underexplored—a critical gap as industry adopts decoder-only architectures.

Defense Methods: Defending against code model attacks presents challenges due to code's discrete nature and functionality requirements. Zhou et al. Zhou et al. (2019) enhanced adversarial training with code property graphs focusing on semantic features. Rabin et al. Rabin et al. (2021) investigated code canonicalization to mitigate syntactic variations. Zou et al. Zou et al. (2023) proposed structure-aware universal defenses. However, existing approaches target specific attack vectors rather than fundamental vulnerability patterns enabling cross-model transferability, highlighting the need for strategies addressing common weaknesses across architectures.

3. Method

3.1. Overview

In this framework, we propose HABITAT, a hierarchical, encoder-guided adversarial attack framework designed to exploit the transferability across SCMs of varying architectures. Our method operates in four stages:

Position-Aware Transformation Discovery (PATD). We reformulate adversarial code modification as a two-level contextual bandit problem. At the global level, we select insertion positions based on semantic salience, while at the local level, we select from six types of syntax-preserving code transformations, guided by position-specific bandits.

Transferable Memory Construction. For each successful attack, we record not only the best transformation strategies but also their contextual reward distributions, forming a transferable "attack memory" for later reuse.

Preference-Guided Strategy Adaptation (PGSA). During PGSA execution, we first check whether there is a stored memory (preference) for each code snippet. If such a memory exists, we adapt it to guide the generation of the adversarial example. Otherwise, we revert to the PATD process.

Adaptive Multi-Model Memory Transfer (MMMT). To enhance cross- architecture generalization, we introduce a model-selection bandit that dynamically determines which previous model's memory is most applicable at each position for current victim SCM. This results in a per-position attack plan tailored to the victim SCM's latent vulnerability surface.

3.2. Threat Scenario Formulation

Our threat model formalizes adversarial capabilities within the context of victim-agnostic code models.

3.2.1. Attack Objectives

We focus on targeted adversarial perturbations against SCMs that satisfy the dual constraints of misclassification and semantic preservation:

$$C(x_{adv}) = y_{adv} \neq y_{true} \wedge S(x_{adv}) \equiv S(x) \tag{1}$$

Where $C(\cdot)$ is the victim classifier, $S(\cdot)$ denotes semantic equivalence, and y_{true} is the correct label. This formulation specifically targets security-critical contexts where misclassifying vulnerable code as secure presents substantial risks to downstream systems—precisely the scenario where robustness guarantees are most essential yet frequently under-evaluated in practice.

3.2.2. Motivation and Adversary's Capability

As defensive mechanisms for SCMs grow increasingly sophisticated and LLM4Code achieves tremendous progress in code understanding and generation, real-world adversaries now predominantly operate under limited knowledge conditions. Modern attackers rarely possess comprehensive information about victim models' parameters, architectures, or downstream tasks, creating a significant disconnect between theoretical transferability concepts and their actual implementation in adversarial campaigns targeting code models.

Our research bridges this gap by generating powerful and practical adversarial examples that effectively affect LLM4Code decisions using only traditional SCM encoders. We systematically exploit the inherent transferability between SCMs, revealing how structural and semantic patterns in code representations create persistent vulnerability surfaces across disparate model architectures and training domains. We impose the most restrictive constraints to date: our attacks require no test-set sampling, no model parameters, purely leveraging traditional SCM encoders' feature space guidance. By reformulating transfer-based attacks as a multi-hierarchical Contextual Bandit Problem, our framework achieves victim-agnostic transferability with no dependency on target model feedback and theoretical convergence guarantees, establishing a framework that accurately reflects real-world adversarial scenarios where attack opportunities remain abundant through strategically crafted transferable examples.

3.3. Hierarchical MAB for Position-Aware Code Transformation Discovery

Based on this threat model, we formulate PATD as a two-level decision problem, and design a hierarchical contextual bandit framework to select optimal insertion points and code transformations without feedback from the victim SCM.

Our architecture consists a two levels of MAB instances:

Global Tier: UCB1 algorithm explores optimal perturbation strategies across model architectures.

Local Tier: Position-sensitive MAB selects attack locations based on code semantic importance.

This hierarchy is formalized as follows:

Let $P = \{p_1, p_2, ..., p_m\}$ be the set of m important positions in the code, and $T = \{t_1, t_2, ..., t_n\}$ be the set of n available code transformations. For each position $p_i \in P$, we maintain a separate MAB instance M_i that learns the effectiveness of each transformation $t_j \in T$ when applied at position p_i.

3.3.1. MAB Principles

Multi-armed bandits (MAB) is a simple but very powerful framework for algorithms that make decisions over time under uncertainty Slivkins et al. (2019). Its core is to learn the value of each arm (code conversion operation) through feedback (attack succeed or not) and balance the relationship between choosing known high-value actions (exploiting) and trying new actions (exploring).

Our work based on the UCB1 (Upper Confidence Bound) algorithm Kuleshov and Precup (2014), which decision depends on:

$$UCB(a) = Q(a) + \alpha \cdot \sqrt{\frac{2 \cdot \ln(t)}{N(a)}} \tag{2}$$

Where:

- $Q(a)$: The estimated value of arm a (exploitation term)
- α: Exploration parameter that controls the degree of exploration
- t: Total number of pulls across all arms
- $N(a)$: Number of times arm a has been pulled

This formula consists of two key components that reveal the essence of the UCB1 algorithm:

- The first term ($Q(a)$) favors selecting actions that have performed best historically (exploitation)
- The second term ($\alpha \cdot \sqrt{\frac{2 \cdot \ln(t)}{N(a)}}$) encourages trying actions that have been selected less frequently (exploration)

As Table 1 had shown, in this work, we introduced 6 different types of code insertion.

3.3.2. Position-specific MAB

For each code snippet, We first select the important positions, which contains a two-stage analysis:

Safe Position Identification: We first analyze the source code to identify syntactically safe insertion points that preserve program structure:

$$\mathcal{P}_{\text{safe}} = \{p : \neg\phi_{\text{syntax}}(p) \wedge \neg\phi_{\text{scope}}(p)\} \tag{3}$$

where ϕ_{syntax} and ϕ_{scope} are predicates that detect syntactic violations and scope conflicts respectively. Our algorithm excludes:

- Multi-line string and comment regions
- Import statements and function/class declarations
- Decorator lines and indentation-sensitive positions

Semantic Importance Ranking: From the safe positions, we select the top-k most semantically important locations by computing feature-space impact:

$$\text{Importance}(p) = ||\text{Emb}(x) - \text{Emb}(x \oplus \text{mask}_p)||_2 \tag{4}$$

Each position-specific MAB, denoted as M_i, encapsulates:

- A unique position identifier p_i
- An internal UCB1 MAB instance with n arms (one per transformation)
- State variables tracking the historically best-performing transformation and its associated reward

Table 1: Code Transformations in Python

Label	Content (in Python)
C1	Insert unreachable code with random variables:`if False: unused_var_1234 = 'hello world!'`
C2	Insert never-executed loop with random variables:`while False: unused_var_5678 = 'unreachable'; break`
C3	Declare unused random-numbered variables: `unused_var_3456 = False`
C4	Define uncalled function with random naming: `def unused_func_7890(): return None`
C5	Insert non-substantive comments: `# NOTE: This is a comment`
C6	Add empty string print statements: `print("")`

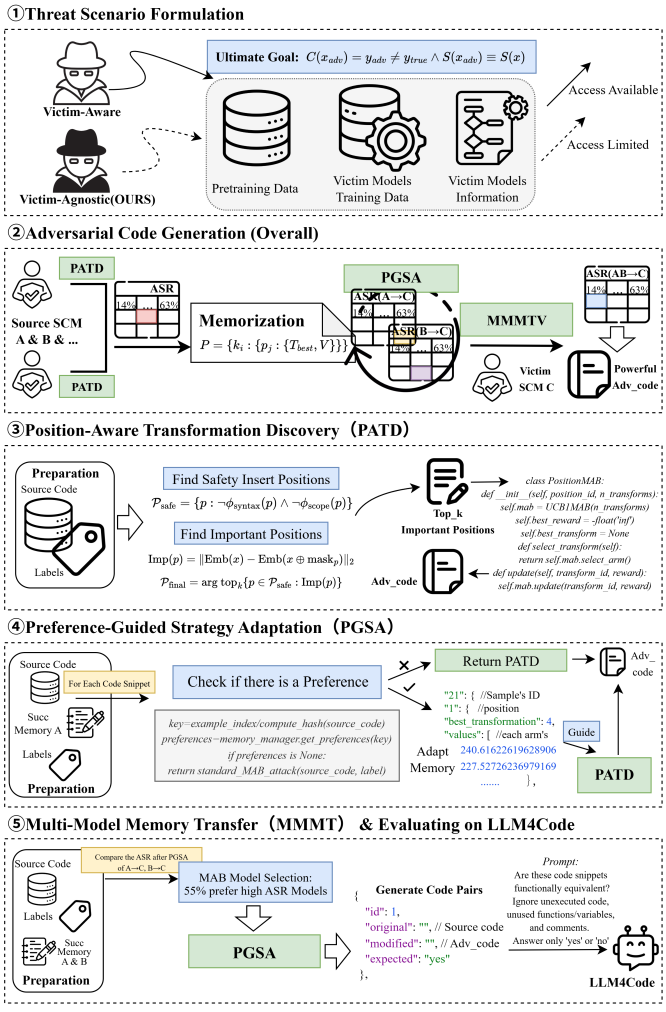


Figure 1: HABITAT Pipeline

Algorithm 1 Position-specific MAB

Initialize: Position identifier p_i, number of transformations n
Requires: MAB object M_i, transformation ID $transformation\_id$, reward $reward$
procedure INITIALIZE(p_i, n)
    M_i.position_id ← p_i
    M_i.mab ← UCB1MAB(n)
    M_i.best_transformation ← null
    M_i.best_reward ← $-\infty$
end procedure
procedure SELECTTRANSFORMATION
    return M_i.mab.select_arm()
end procedure
procedure UPDATE($transformation\_id$, $reward$)
    M_i.mab.update($transformation\_id$, $reward$)
    if $reward > M_i$.best_reward **then**
        M_i.best_reward ← $reward$
        M_i.best_transformation ← $transformation\_id$
    end if
end procedure

The position MAB manages the selection of transformations through a delegation pattern, as illustrated in equation 2.

Critical insight : Our empirical analysis reveals that although there's a considerable transferability between different SCMs, the position importance distributions across different SCMs exhibit surprisingly low correlation. We will discuss this phenomenon particularly in the next section.

This observation, contrary to intuition, actually strengthens our approach: our hierarchical MAB architecture effectively learns position-specific strategies independently, making it robust to the absence of cross-architecture position correlations.

The position-specific MAB overcomes this challenge by:

- Learning optimal strategies independently for each position
- Not relying on pre-assumed position importance correlations
- Adapting to architecture-specific characteristics through online learning

3.4. Memory-Augmented Mechanisms

While the hierarchical bandit enables efficient attack on individual victim models, we further enhance transferability by storing transformation patterns and reward histories from successful attacks, which allowing us to reuse and adapt past strategies across samples.

3.4.1. Persistent Memory Architecture for Attack Transferability

For each successful attack, we store a comprehensive preference profile that captures both the optimal transformation strategies and their associated reward distributions:

$$\mathcal{M}(x) = \{(p_i, t_i^*, \mathbf{V}_i) : \forall p_i \in \mathcal{P}_{\text{important}}\} \tag{5}$$

where p_i denotes the position index, t_i^* represents the optimal transformation for position p_i, and $\mathbf{V}_i \in \mathbb{R}^{|\mathcal{T}|}$ encodes the empirical reward vector for all transformations at position p_i. The reward for each transformation is computed as:

$$r_{t,p} = \|\text{Emb}(x) - \text{Emb}(x \oplus \mathcal{T}_t(p))\|_2 + [\text{Attack}_{\text{success}}(x \oplus \mathcal{T}_t(p))] \tag{6}$$

This composite reward function captures both the semantic perturbation magnitude in the feature space and the actual attack efficacy, enabling nuanced transfer of attack strategies.

Algorithm 2 Adaptive Multi-Model Selection

Input: Source code S, Model experiences E_1, E_2, Positions P
Output: Adversarial code S'
Initialize model selection MABs: $M_{\text{model}}[p] \leftarrow$ UCB1MAB(2) for $p \in P$
Initialize transformation MABs using stored experiences E_1, E_2
for iteration = 1 to max_iterations **do**
    for each position $p \in P$ **do**
        model_idx ← MAB selection (70%) or random (30%)
        transform_idx ← Select transformation using chosen model's MAB
        Store selections: selected_transforms[p], model_choices[p]
    end for
    $S_c \leftarrow$ Apply transformations to S
    reward ← Evaluate attack effectiveness of S_c
    Update transformation and model selection MABs with reward
    if attack succeeds **then return** S_c
    end if
end for
return best adversarial example found

3.5. Adaptive Multi-Model Memory Transfer

To maximize attack effectiveness across different victim SCMs, we propose an Adaptive multi-model approach that intelligently leverages knowledge from multiple previous successful attacks. As illustrated in Figure 1, before MMMT, we first perform a comprehensive pairwise ASR (Attack Success Rate) analysis across source and victim SCMs. This step identifies which individual source SCM produces more transferable adversarial examples for a given target. These ASR values are stored and used as priors to inform the MMMT process, this method combines and adaptively selects from the attack preferences learned from different victim SCMs.

3.5.1. Hierarchical Bandit Structure

We also abstract the problem by using a hierarchical multi-armed bandit structure with two levels of decision-making:

1. **Model Selection Level**: For victim model C and its each position, which model's attack experience provides better guidance? (which one shares more vulnerability with C?)
2. **Transformation Selection Level**: For each code snippet, given the selected model's memory(if there is one), which transformation should be applied?

3.5.2. Adaptive Multi-Model Knowledge Selection

Different victim models often exhibit distinct vulnerabilities at different code positions, making static or uniform knowledge transfer suboptimal. To address this, we propose an adaptive multi-model selection strategy that dynamically chooses which prior SCM's experience to leverage at each important code location. Instead of applying a single SCM's preferences globally or averaging across models, we maintain a position-specific MAB selector that learns, over time, which model offers the most effective guidance at each location. This enables fine-grained adaptation, allowing the attack to draw on Model A's knowledge in some parts of the code and Model B's in others, tailored to the victim SCM's unique behavior. The details are presented in Algorithm 2. Our adaptive model selection process works as follows:

1. For each position, we maintain a model-selection MAB (2 arms: model A, model B)
2. When selecting transformations, we first use the model-selection MAB to decide which model's experience to follow
3. The chosen model's transformation preferences then guide the selection of the specific transformation
4. After applying transformations and evaluating results, we update both:
    - The transformation MAB of the chosen model (lower-level decision)
    - The model-selection MAB (higher-level decision)
5. As iterations progress, each position's model-selection MAB biases toward the model providing better guidance

This approach balances exploration and exploitation by combining MAB-guided selection with randomness in early stages, then relying more heavily on learned preferences in later stages.

4. Analysis of the Dominant Factors on Transferable Vulnerability

Understanding the fundamental mechanisms behind adversarial transferability in SCMs represents a critical yet underexplored frontier in software security research.

The transferability observed in SCMs stems fundamentally from their ability to learn and represent the intrinsic, underlying properties of source code that are common across different tasks and even different model architectures. Pre-training on large, diverse code corpora allows models to capture generalized features and structures of programming languagesHussain et al. (2020). Research indicates that despite variations in design and pre-training objectives, different SCMs are capable of learning similar fundamental code characteristics, including syntactic and semantic informationHan et al. (2021).

In this section, we present the first comprehensive investigation of cross-architecture adversarial transferability in SCMs, systematically identifying and quantifying the underlying mechanisms rather than merely documenting the phenomenon.

4.1. Insert Positions

We selected 100 code snippets from Authorship Attribution, as demonstrated in Figure 2, we quantify this through pairwise correlation analysis:

$$\rho(M_i, M_j) = \text{Cov}(\mathbf{I}_{M_i}, \mathbf{I}_{M_j})/(\sigma_{M_i}\sigma_{M_j}) \tag{7}$$

where $\mathbf{I}_M \in \mathbb{R}^{|\mathcal{P}|}$ represents the importance vector for model M across all positions $\mathcal{P}$. Our experiments show that $\rho(M_i, M_j) < 0.1$ for most model pairs, especially between CodeBERT and UniXcoder, indicating that position sensitivity is model-specific rather than universal, therefore it is not a dominant factor that cause the cross-architecture transferability.

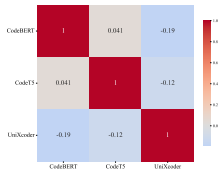


Figure 2: Samples position correlation

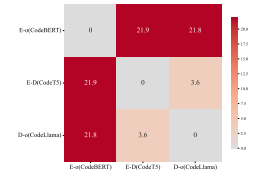


Figure 3: Model feature space similarity

4.2. Success Rates of Different Insertions Across Different SCMs

The statistical distributions in Figure 4 represent data from successfully attacked samples across different SCMs. Each model exhibits distinct vulnerability patterns to code transformation types (Table 1). C1 and C2 represent common vulnerabilities across all models with detailed differences: CodeBERT shows balanced vulnerability across six insertion types, UniXcoder exhibits strong bias toward C2 (52.7%) while resisting C5-C6 transformations, and CodeT5 demonstrates particular weakness to C1 and C4 (34.6% and 38.3% respectively). These distinct vulnerability patterns explain their limitations

	C1	C2	C3	C4	C5	C6
UniXcoder	31.2	52.7	8.6	7.5	0.0	0.0
CodeBERT	15.1	20.4	7.5	12.9	21.5	22.6
CodeT5	34.6	14.8	6.2	38.3	2.5	3.7
Average	**27.0**	**29.3**	7.4	19.6	8.0	8.8

Figure 4: Successful samples average insertion correlation

across different programming contexts, enabling our approach to maximize transferability by leveraging each model's unique patterns rather than treating them as uniform systems. While SCMs learn similar underlying code properties, they vary significantly in representation mastery Ma et al. (2024)Karmakar and Robbes (2021). Different models show varying strengths in code syntax and semantics, with CodeT5 and CodeBERT excelling at capturing control flow and data flow dependencies compared to UniXcoder Niu et al. (2023)Xiao et al. (2023). These differences stem from variations in architecture, pre-training objectives, and training data, leading to distinct internal representations and differential responses to adversarial perturbations. The existence of common vulnerability types (C1 and C2) provides transferability pathways, making insertion type a dominant factor in cross-architecture adversarial transferability.

4.3. Local via Global Attentions Distribution

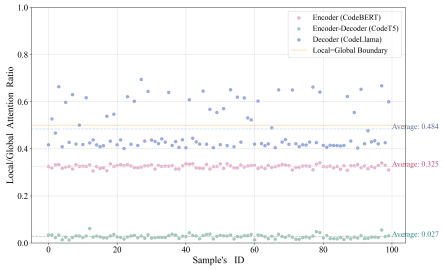


Figure 5: Different SCMs Local/Global attention distribution

Attention mechanisms Vaswani et al. (2017) play a crucial role in how different code models process and understand code structure.

We quantified the Local/Global Attention Ratio ($\mathcal{R}_{LG}$) across 100 code samples from Authorship Attribution for three representative model architectures. For each model, we define the ratio as:

$$\mathcal{R}_{LG} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbf{A}_{ij}\cdot\mathbf{1}(|i-j|\leq\delta)}{\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbf{A}_{ij}} \tag{8}$$

where $\mathbf{A}_{ij}$ represents the attention weight from token i to token j, n is the sequence length, $\delta = 5$ is our local context window, and $\mathbf{1}(\cdot)$ is the indicator function that equals 1 when the condition is true and 0 otherwise.

Our results show significant architectural differences. Decoder-only models (CodeLlama) maintain the highest Local/Global Attention Ratio with an average of $\mathcal{R}_{LG} = 0.484$, indicating they allocate nearly equal attention to both local & global code contexts. Encoder-only models (CodeBERT) show a moderate but consistent ratio averaging $\mathcal{R}_{LG} = 0.325$, while Encoder-Decoder models (CodeT5) exhibit a dramatically lower ratio of $\mathcal{R}_{LG} = 0.027$, indicating an overwhelming preference for global contextual understanding.

This analysis provides a direct explanation for the model-specific vulnerability patterns observed in our experiments and suggests that attention distribution characteristics represent a fundamental factor influencing adversarial transferability across different code model architectures.

4.4. Length, Logical Structure and Complexity of Code Snippet

Code complexity presents a multifaceted characteristic that potentially influences the cross-architecture transferability of adversarial examples. We systematically investigate this relationship through a multidimensional analysis of code structural properties and their impact on representation similarity across different architectures.

4.4.1. Complexity Quantification Framework

We define code complexity through a six-dimensional metric system, formalizing each dimension as follows:

1. **Line Count** (L_c): Raw quantity of non-empty code lines
2. **Character Count** (C_c): Total number of characters in the source code
3. **Maximum Indentation Level** (I_{max}): Maximum nesting depth, where indentation level is calculated as:

$$I(line) = \text{leading whitespace count}/4 \tag{9}$$

4. **Control Structure Count** (CS_c): Frequency of control flow constructs:

$$CS_c = \sum_{line \in L_c} \mathbf{1}_{\exists k \in K : k \in line} \tag{10}$$

where K = {if, else, elif, for, while, try, except, with, def, class}

5. **Composite Complexity Score** (C_{score}): An aggregated metric synthesizing multiple dimensions:

$$C_{score} = L_c/10 + I_{max} + CS_c/5 \tag{11}$$

For comparative analysis, we categorize samples as "High" or "Low" complexity based on their position relative to the median C_{score} across the dataset:

$$\text{Complexity Category}(s) = \begin{cases} \text{High}, & \text{if } C_{score}(s) > \text{median}(C_{score}) \\ \text{Low}, & \text{otherwise} \end{cases} \tag{12}$$

4.4.2. Cross-Architecture Similarity Analysis

For each architecture pair (M_i, M_j), we calculate the cosine similarity between their feature representations:

$$\text{Sim}(M_i(s), M_j(s)) = M_i(s) \cdot M_j(s) / \|M_i(s)\| \cdot \|M_j(s)\| \tag{13}$$

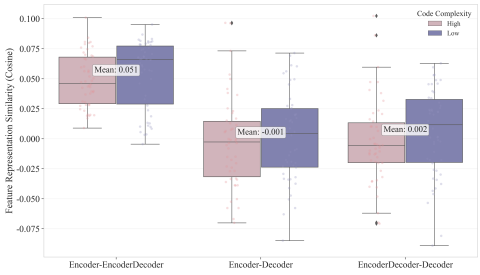


Figure 6: Feature representation similarity across different model architecture pairs categorized by code complexity.

where $M_i(s)$ represents the feature vector generated by model M_i for code sample s.

To investigate how code complexity affects cross-architecture transferability, we calculate separate mean similarity values (μ) for high and low complexity subsets:

$$\mu^{High}_{M_i,M_j} = \frac{1}{|S_{High}|} \sum_{s \in S_{High}} \text{Sim}(M_i(s), M_j(s)) \tag{14}$$

$$\mu^{Low}_{M_i,M_j} = \frac{1}{|S_{Low}|} \sum_{s \in S_{Low}} \text{Sim}(M_i(s), M_j(s)) \tag{15}$$

where $S_{High} = \{s : \text{Complexity Category}(s) = \text{High}\}$ and $S_{Low} = \{s : \text{Complexity Category}(s) = \text{Low}\}$ are the high and low complexity sample sets respectively, determined by Equation (5). The overall mean similarity is:

$$\mu_{M_i,M_j} = \frac{|S_{High}| \cdot \mu^{High}_{M_i,M_j} + |S_{Low}| \cdot \mu^{Low}_{M_i,M_j}}{|S_{High}| + |S_{Low}|} \tag{16}$$

This stratified analysis allows us to quantify whether code complexity systematically influences the consistency of feature representations across different model architectures.

4.4.3. Results and Analysis

Each data point in Figure 6 represents a code sample, with similarity values computed via Equation 11. The figure reveals key patterns in how code complexity affects cross-architecture feature similarity:

1. **Architecture-Dependent Similarity**: Encoder-Encoder Decoder pairs show substantially higher similarity ($\mu = 0.051$) than other combinations ($\mu \leq 0.002$), indicating architectural compatibility is the primary factor in representation similarity.
2. **Variable Complexity Effects**: Code complexity impacts similarity differently across architecture pairs, with Encoder-EncoderDecoder favoring low-complexity code while Encoder-Decoder shows opposite trends.
3. **Structural Outliers**: All pairs exhibit outlier samples with exceptional similarity values, indicating specific code structures maintain consistent representations across architectures regardless of complexity.

While previous work Pierazzi et al. (2020) emphasized complex and redundant code for bypassing detection, our find-

ings suggest that specific code structures with consistent cross-architecture representations are more effective than general code complexity. Although complexity is not the dominant factor in adversarial transferability, high-similarity outliers across architecture pairs indicate that certain samples can achieve notable cross-model transferability, with architectural compatibility as the primary determinant and specific structural patterns as secondary but significant factors.

4.5. Semantic Comprehension Disparities

To further understand the underlying mechanisms driving adversarial transferability across different model architectures, we analyze the feature distance between models using cosine similarity metrics. Figure 3 presents a comprehensive distance matrix computed in our three above-mentioned target models.

The distance analysis reveals distinct clustering patterns among different architectural paradigms. Notably, CodeT5 and CodeLlama demonstrate the smallest inter-model distance (3.559), suggesting that these models operate in relatively similar feature spaces despite their architectural differences. This proximity in the feature space partially explains the higher transferability rates observed between encoder-decoder and decoder-only models in our empirical evaluations. This alignment can be theoretically supported by Fu et al. Fu et al. (2023), who demonstrate that decoder-only language models can be interpreted as regularized encoder-decoder models, where the decoder component shares many functional similarities in how they process and generate sequences.

The distance values are computed using the Euclidean distance between feature vectors extracted from each model on a shared set of code samples:

$$D(M_i, M_j) = \frac{1}{|S|} \sum_{s \in S} \sqrt{\sum_{k=1}^{\min(|\mathbf{f}i,s|,|\mathbf{f}j,s|)} (f_{i,s,k} - f_{j,s,k})^2} \qquad (17)$$

where $\mathbf{f}_{i,s}$ represents the feature vector of sample s in model M_i, and S is the set of common samples across all models.

In contrast, CodeBERT exhibits significantly larger distances from both CodeT5 (21.945) and CodeLlama (21.846), with relatively uniform distances suggesting that the architectural gap between encoder-only and generation-capable models represents a more significant barrier to transferability than differences between encoder-decoder and decoder-only designs. These metrics provide quantitative evidence that models with similar training objectives and architectural components are more susceptible to shared vulnerabilities, facilitating adversarial transferability across model boundaries.

5. Evaluation

5.1. Victim SCMs Training

In this study, we employ three state-of-the-art traditional SCMs as victim models: CodeBERT Feng et al. (2020), CodeT5 Wang et al. (2021), UniXcoder Guo et al. (2022). We selected two code classification datasets, the Authorship AttributionAlsulami et al. (2017) and the Defeat DetectionDevlin et al. (2019), the former originates from the Google Code Jam challenge and aims to identify the author of a given code snippet, while the latter is a C dataset that combines two popular open-sourced projects: FFmpeg3 and Qemu4.

Table 2 shows the training accuracy of our victim SCMs on clean datasets before adversarial attacks.

Table 2: Clean Accuracy Comparison of Different Models on Various Datasets

Dataset	Model	Clean Acc
Authorship Attribution	CodeBERT	74.24
	CodeT5	77.27
	UniXCoder	78.03
Defeat Detection	CodeBERT	57.54
	CodeT5	58.97
	UniXCoder	44.14

5.2. Important Top_k Position Chosen

To determine the optimal number of insertion positions for PATD, we conduct a position selection experiment across different values of k to balance attack effectiveness with code perturbation constraints.

5.2.1. Experimental Setup

We evaluate the impact of varying $k \in \{1, 2, 3, 5, 8, 10, 15\}$ across CodeBERT, CodeT5, and UniXcoder using three key metrics: **Attack Success Rate (ASR):** Proportion of samples whose predictions are successfully changed. **Feature Distance (FD):** Semantic distance between original and adversarial code representations, equivalent to equation 4. **Code Modification Rate (CMR):** Percentage of code tokens modified during attack.

5.2.2. Position Selection Strategy

Table 3: ASR Comparison of Different Models with PATD and Baselines on Authorship Attribution

Model	CodeTAE	PATD(ours)	Random-only(ours)
CodeBERT	11.24(+14.29)	25.53	15.96(+9.57)
CodeT5	5.88(+16.67)	22.55	15.00(+7.55)
UniXcoder	15.38(+14.29)	29.67	19.78(+9.89)
Avg	10.83(+15.09)	25.92	16.88(+9.04)

Table 4: ASR Comparison of Different Models with PATD and Baselines on Defeat Detection

Model	CodeTAE	PATD(ours)	Random-only(ours)
CodeBERT	5.86(+8.6)	10.46	3.35(+7.11)
CodeT5	17.65(+22.99)	40.64	19.25(+21.39)
UniXcoder	8.23(+11.39)	19.62	12.03(+7.59)
Avg	10.58(+12.99)	23.57	11.54(+12.03)

We evaluate PATD from two perspectives: attack effectiveness and stealthiness. Higher k values improve ASR but increase Code Modification Rate and Feature Distance, compromising stealthiness and increasing detection risk. Therefore,

we seek optimal k that maximizes ASR while minimizing code modifications and semantic distance.

5.2.3. Results and Analysis

We select $k = 3$ as the optimal configuration for both datasets based on the trade-off between attack effectiveness and stealthiness. Detailed ASR, FD and CMR results for all tested k configurations are presented in Table 5 and 6.

Table 5: SCM FD & CMR & ASR [%] Comparison Across Different Top-k Positions on Authorship Attribution

Models	k=1	k=2	k=3	k=5	k=8	k=10	k=15
CodeBERT	217.1	218.6	221.4	226.7	224.3	225.6	226.5
	9.9	18.9	28.3	48.7	74.4	88.9	113.0
	24.47	30.85	32.98	32.98	36.17	37.23	41.49
CodeT5	130.3	154.9	141.7	179.3	253.5	375.9	1591.8
	7.9	15.5	22.1	39.2	59.5	75.1	104.7
	24.51	24.51	24.51	29.41	39.22	46.08	57.84
UniXcoder	352.0	480.6	457.9	486.3	509.0	468.0	502.5
	26.0	46.7	64.3	89.7	98.8	98.7	99.3
	20.88	29.67	29.67	40.66	50.55	51.65	50.55

Table 6: SCM FD & CMR [%] & ASR [%] Comparison Across Different Top-k Positions on Defeat Detection

Models	k=1	k=2	k=3	k=5	k=8	k=10	k=15
CodeBERT	1.31	1.74	2.07	2.56	3.00	3.17	3.42
	2.5	4.4	6.3	9.4	13.4	15.5	19.3
	8.37	7.11	11.72	11.30	14.64	15.48	15.90
CodeT5	0.85	1.10	1.37	1.63	2.08	2.05	2.64
	1.9	3.3	4.9	6.9	10.0	11.2	15.9
	36.90	39.57	41.48	45.45	47.06	49.73	44.92
UniXcoder	5.17	6.57	7.68	10.03	12.34	13.54	15.21
	2.3	4.0	5.6	8.8	12.3	14.3	17.5
	13.29	18.35	21.52	19.62	24.05	23.42	24.68

5.3. Attack Effectiveness Across Exploration Rates

We design experiments that systematically explore the trade-off between exploration and exploitation in multi-model scenarios.

5.3.1. Baseline

We adopt CodeTAE Yang et al. (2024) as our baseline, and we also designed a random-only baseline for the PATD method that uses only random combinations without the MAB exploration-exploitation process.

5.3.2. Evaluation Metric

The effectiveness of the attack is quantified by the **Attack Success Rate (ASR)**, defined as the fraction of initially correctly classified samples that are misclassified after the attack:

$$\text{ASR} = \frac{|\mathcal{S}_{\text{success}}|}{|\mathcal{S}_{\text{correct}}|}$$

where $\mathcal{S}_{\text{correct}}$ denotes the set of samples correctly classified by the victim model prior to the attack, and $\mathcal{S}_{\text{success}} \subseteq \mathcal{S}_{\text{correct}}$ is the subset that is misclassified post-attack.

5.3.3. Experimental Parameters

We systematically vary the exploration rate from 14% to 63% in 7% increments to understand how guidance levels influence attack effectiveness. This rate equals MAB exploration rate $\times$ 0.7, with 70% MAB-directed exploitation and 30% random exploration. Lower rates emphasize exploration, while higher rates focus on exploitation.

Table 7: ASR (%) Results of CodeBERT with Different Exploration Rates and Strategies on Defeat Detection

Strategy	14%	21%	28%	35%	42%	49%	56%	63%	Avg
Original	11.72	10.46	11.72	13.39	12.55	11.30	14.64	13.81	12.45
+CodeT5	14.64	10.04	10.46	12.13	12.55	12.13	15.90	13.81	12.71(+0.26)
FEM	12.97	10.88	11.72	12.55	12.55	11.72	15.06	15.06	12.81(+0.36)
+UniX	12.13	9.62	12.13	12.13	13.39	11.72	12.97	12.55	12.08(-0.37)
FEM	13.39	9.21	13.39	12.97	13.81	10.46	15.48	14.23	12.87(+0.42)
+BOTH	11.72	10.88	10.04	15.48	10.46	11.30	15.48	14.64	12.50(+0.05)

Table 8: ASR (%) Results of CodeT5 with Different Exploration Rates and Strategies on Defeat Detection

Strategy	14%	21%	28%	35%	42%	49%	56%	63%	Avg
Original	41.48	40.64	40.64	40.64	42.25	41.18	42.78	43.85	41.68
+CodeBE	41.18	40.64	40.64	40.64	40.64	38.50	40.64	44.39	40.91(-0.77)
FEM	33.69	34.22	33.69	33.69	32.09	32.62	35.29	35.29	33.82(-7.86)
+UniX	39.57	37.97	37.97	22.78	42.25	37.97	37.97	42.25	37.34(-4.34)
FEM	23.53	26.74	27.27	27.27	28.88	29.95	36.36	37.97	29.75(-11.93)
+BOTH	43.85	40.11	40.11	40.11	41.18	40.11	45.99	45.99	42.18(+0.50)

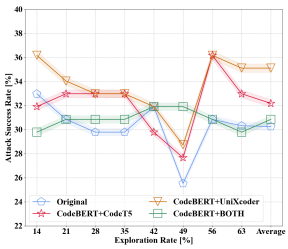
Table 9: ASR (%) Results of UniXcoder with Different Exploration Rates and Strategies on Defeat Detection

Strategy	14%	21%	28%	35%	42%	49%	56%	63%	Avg
Original	21.52	20.25	21.52	20.89	20.89	19.62	19.62	20.25	20.57
+CodeBE	18.35	22.78	22.15	21.52	18.99	17.09	18.99	19.62	20.06(-0.51)
FEM	21.52	22.78	22.78	20.89	19.62	17.72	21.52	20.89	20.97(+0.40)
+CodeT5	18.99	21.52	22.78	18.35	18.35	15.82	17.09	20.25	19.14(-1.43)
FEM	17.09	19.62	24.05	19.62	18.35	18.35	17.72	19.62	19.30(-1.27)
+BOTH	21.52	19.62	20.89	18.35	20.25	18.35	19.62	19.62	19.94(-0.63)

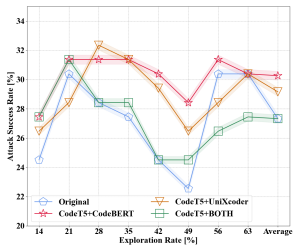
5.3.4. Result Analysis

Comparison With Baseline: Our PATD method demonstrates substantial improvements over both baselines across all models and tasks. For Authorship Attribution, PATD achieves average improvements of 15.09% over CodeTAE and 9.04% over random baseline, with consistent gains across CodeBERT (14.29%), CodeT5 (16.67%), and UniXCoder (14.29%). On Defeat Detection, PATD shows even more pronounced advantages with 13.0% and 11.03% improvements respectively, with CodeT5 achieving the most significant improvement of 22.99% over CodeTAE.

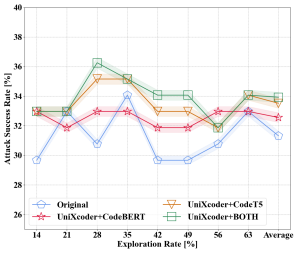
PGSA vs. MMMT Performance: Our dual auxiliary model strategy (MMMT) consistently outperforms single auxiliary model approach (PGSA) across different exploration rates. CodeBERT with UniXcoder achieves superior performance with peak ASR of 35-36%, while CodeT5 benefits from CodeT5 and UniXcoder pairing, reaching approximately 32%



(a) ASR Comparison of CodeBERT through PATD, PGSA, MMMT



(b) ASR Comparison of CodeT5 through PATD, PGSA, MMMT



(c) ASR Comparison of UniXcoder through PATD, PGSA, MMMT

Figure 7: Comparison of different traditional SCMs on Authorship Attribution

ASR. UniXcoder shows the most dramatic improvement with MMMT, achieving 34-35% ASR at optimal exploration rates. MMMT also exhibits more stable performance across varying exploration rates.

Full Experience Memory (FEM) Analysis: For Defeat Detection, FEM implementation demonstrates the value of leveraging complete experience over success-only memory. CodeBERT shows consistent modest improvements with CodeT5 (+0.36%) and UniXcoder (+0.42%) knowledge transfers, while UniXcoder achieves positive gains with CodeBERT knowledge (+0.40%). These findings highlight that incorporating both successful and failed attack experiences enhances transferability compared to using only successful patterns.

5.3.5. Discussions

Memorization Pattern Complementarity: Single-guide strategies often outperform dual-guide approaches for Authorship Attribution, indicating that memorization patterns across SCMs are not always complementary. When models provide conflicting signals, the adaptive mechanism struggles to synthesize information effectively.

Exploration-Exploitation Balance: Performance stability across varying exploration rates validates our hybrid approach with 30% random exploration, preventing over-dependence on auxiliary model insights while maintaining discovery of novel vulnerabilities.

Security Implications: Cross-model knowledge transfer reveals that memorization patterns from one model can enhance attacks against others. Model providers should evaluate vulnerabilities within the broader ecosystem, accounting for cross-model information leakage.

Adaptive Strategy Refinement: Mixed results from multi-model guidance suggest naive combination of knowledge sources may introduce interference. Future work should explore sophisticated fusion mechanisms like attention-based guidance weighting or dynamic model selection.

5.4. Evaluating on LLM4Code

5.4.1. Preparation

We evaluate transferability on three representative LLM4Code: CodeLlama-7B Roziere et al. (2023), StarCoderLi et al. (2023), and StarCoder2-3B Lozhkov et al. (2024). CodeLlama-7B specializes in code understanding and generation with strong reasoning capabilities, StarCoder provides robust code completion and generation across diverse programming languages, while StarCoder2-3B offers efficient code processing with broad language support.

Table 10: ASR (%) Comparison on Authorship Attribution Across Different LLM4Code

Source SCMs	UniXcoder			CodeBERT			CodeT5		
	ASR	Recall	F1	ASR	Recall	F1	ASR	Recall	F1
CodeLlama-7B	28.41	71.59	83.44	16.84	83.16	90.80	41.58	58.42	73.75
StarCoder	6.72	93.18	96.47	9.47	90.53	95.03	9.90	91.00	95.29
StarCoder2-3B	7.95	92.05	95.86	15.79	84.21	91.43	6.93	93.07	96.41

Table 11: ASR (%) Comparison on Defeat Detection Across Different LLM4Code

Source SCMs	UniXcoder			CodeBERT			CodeT5		
	ASR	Recall	F1	ASR	Recall	F1	ASR	Recall	F1
CodeLlama-7B	64.56	35.44	52.34	57.74	42.26	59.41	55.61	44.39	61.48
StarCoder	3.16	96.84	98.39	5.86	94.14	96.98	9.09	90.91	95.51
StarCoder2-3B	15.82	84.18	91.41	23.43	76.57	86.73	25.67	74.33	85.28

After completing PGSA and MMMT, the victim SCM collects all ASR results, selects the best-performing parameters, and generates a JSON file containing both original and adversarial code. This file is then used to queries the LLM4Code above, instructing it to determine if two code snippets are functionally equivalent while ignoring non-executed code (like if false), unused functions/variables, and comments.

We calculate key performance metrics:

$$\text{Precision: } \mathcal{P}_{rec} = TP/(TP + FP) \tag{18}$$

$$\text{ASR: } \mathcal{ASR} = 1 - \mathcal{P}_{rec} \tag{19}$$

$$\text{Recall: } \mathcal{R} = TP/(TP + FN) \tag{20}$$

$$\text{F1-Score: } \mathcal{F}_1 = 2 \cdot (\mathcal{P}_{rec} \cdot \mathcal{R})/(\mathcal{P}_{rec} + \mathcal{R}) \tag{21}$$

5.4.2. Result and Analysis

As shown in Tab 10 and Tab 11, our evaluation reveals distinctive vulnerability patterns across LLM4Code models. CodeLlama-7B shows the highest susceptibility with ASRs reaching 41.58% on Authorship Attribution and 64.56% on Defeat Detection, suggesting significant transferability from traditional SCMs. StarCoder demonstrates exceptional robust-

ness, maintaining low ASRs (3.16%-9.90%) and high F1 scores (95.03%-98.39%) across both datasets. StarCoder2-3B exhibits moderate vulnerability, showing higher ASRs on Defeat Detection (15.82%-25.67%) compared to Authorship Attribution (6.93%-15.79%). These findings highlight that adversarial transferability varies substantially based on model architecture and dataset characteristics.

5.4.3. Discussions

Our study reveals significant variation in adversarial robustness among LLM4Code models. CodeLlama-7B's high vulnerability contrasts with StarCoder's exceptional resilience, suggesting fundamental architectural differences that impact security. Higher attack transferability on Defeat Detection versus Authorship Attribution indicates that task characteristics substantially influence vulnerability, suggesting security evaluations should prioritize task-specific testing. These findings highlight a critical security-capability tradeoff: as models scale to enhance performance, they may simultaneously increase vulnerability to adversarial attacks. Future work should investigate architectural elements contributing to StarCoder's robustness and develop transferability-aware training techniques.

6. Conclusion

This work presents the first systematic investigation of transferable adversarial vulnerabilities between traditional SCMs and LLM4Code systems. Our HABITAT method achieves up to 64% attack success rates against LLM4Code using only traditional SCM guidance, significantly outperforming existing methods. Through comprehensive analysis of five factors governing cross-architecture vulnerability transfer, we reveal that fundamental security weaknesses persist across the entire SCM ecosystem despite architectural diversity, enabling cascading security failures. Our findings provide essential insights for developing robust adversarial defenses and proactively strengthening SCM systems against cross-architecture attacks.

References

Alon, U., Zilberstein, M., Levy, O., Yahav, E., 2019. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3, 1–29.

Alsulami, B., Dauber, E., Harang, R., Mancoridis, S., Greenstadt, R., 2017. Source code authorship attribution using long short-term memory based networks, in: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22, Springer. pp. 65–82.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al., 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 .

Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L.N., Poshyvanyk, D., Monperrus, M., 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering 47, 1943–1959.

Devlin, J., Chang, M.W., Lee, K., Toutanova, K., 2019. Bert: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers), pp. 4171–4186.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 .

Fu, Z., Lam, W., Yu, Q., So, A.M.C., Hu, S., Liu, Z., Collier, N., 2023. Decoder-only or encoder-decoder? interpreting language model as a regularized encoder-decoder. arXiv preprint arXiv:2304.04052 .

Gu, X., Zhang, H., Kim, S., 2018. Deep code search, in: Proceedings of the 40th international conference on software engineering, pp. 933–944.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., Yin, J., 2022. Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850 .

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 .

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al., 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:2401.14196 .

Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., Qiu, J., Yao, Y., Zhang, A., Zhang, L., et al., 2021. Pre-trained models: Past, present and future. AI Open 2, 225–250.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2018. Deep code comment generation, in: Proceedings of the 26th conference on program comprehension, pp. 200–210.

Hussain, Y., Huang, Z., Zhou, Y., Wang, S., 2020. Deep transfer learning for source code modeling. International Journal of Software Engineering and Knowledge Engineering 30, 649–668.

Karmakar, A., Robbes, R., 2021. What do pre-trained code models know about code?, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 1332–1336.

Kuleshov, V., Precup, D., 2014. Algorithms for multi-armed bandit problems. arXiv preprint arXiv:1402.6028 .

Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al., 2023. Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161 .

Liu, Y., Chen, X., Liu, C., Song, D., 2016. Delving into transferable adversarial examples and black-box attacks. arXiv preprint arXiv:1611.02770 .

Lozhkov, A., Li, R., Allal, L.B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y., et al., 2024. Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173 .

Lutellier, T., Pham, H.V., Pang, L., Li, Y., Wei, M., Tan, L., 2020. Coconut: combining context-aware neural translation models using ensemble for program repair, in: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp. 101–114.

Ma, W., Liu, S., Zhao, M., Xie, X., Wang, W., Hu, Q., Zhang, J., Liu, Y., 2024. Unveiling code pre-trained models: Investigating syntax and semantics capacities. ACM Transactions on Software Engineering and Methodology 33, 1–29.

Nguyen, T.D., Zhou, Y., Le, X.B.D., Thongtanunam, P., Lo, D., 2023. Adversarial attacks on code models with discriminative graph patterns. arXiv preprint arXiv:2308.11161 .

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C., 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 .

Niu, C., Li, C., Ng, V., Chen, D., Ge, J., Luo, B., 2023. An empirical comparison of pre-trained models of source code, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE. pp. 2136–2148.

Papernot, N., McDaniel, P., Goodfellow, I., 2016. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. arXiv preprint arXiv:1605.07277 .

Pierazzi, F., Pendlebury, F., Cortellazzi, J., Cavallaro, L., 2020. Intriguing properties of adversarial ml attacks in the problem space, in: 2020 IEEE symposium on security and privacy (SP), IEEE. pp. 1332–1349.

Rabin, M.R.I., Bui, N.D., Wang, K., Yu, Y., Jiang, L., Alipour, M.A., 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. Information and Software Technology 135, 106552.

Ramakrishnan, G., Albarghouthi, A., 2022. Backdoors in neural models of source code, in: 2022 26th International Conference on Pattern Recognition (ICPR), IEEE. pp. 2892–2899.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., et al., 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950 .

Slivkins, A., et al., 2019. Introduction to multi-armed bandits. Foundations and Trends® in Machine Learning 12, 1–286.

Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M.M., 2014. Towards a big data curated benchmark of inter-project code clones, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE. pp. 476–480.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al., 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288 .

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems 30.

Wan, Y., Bi, Z., He, Y., Zhang, J., Zhang, H., Sui, Y., Xu, G., Jin, H., Yu, P., 2024. Deep learning for code intelligence: Survey, benchmark and toolkit. ACM Computing Surveys 56, 1–41.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 .

Xiao, Y., Zuo, X., Xue, L., Wang, K., Dong, J.S., Beschastnikh, I., 2023. Empirical study on transformer-based techniques for software engineering. arXiv preprint arXiv:2310.00399 .

Yang, Y., Fan, H., Lin, C., Li, Q., Zhao, Z., Shen, C., 2024. Exploiting the adversarial example vulnerability of transfer learning of source code. IEEE Transactions on Information Forensics and Security .

Yang, Z., Shi, J., He, J., Lo, D., 2022. Natural attack for pre-trained models of code, in: Proceedings of the 44th International Conference on Software Engineering, pp. 1482–1493.

Yefet, N., Alon, U., Yahav, E., 2020. Adversarial examples for models of code. Proceedings of the ACM on Programming Languages 4, 1–30.

Zhang, H., Li, Z., Li, G., Ma, L., Liu, Y., Jin, Z., 2020. Generating adversarial examples for holding robustness of source code processing models, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 1169–1176.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE. pp. 783–794.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems 32.

Zou, A., Wang, Z., Carlini, N., Nasr, M., Kolter, J.Z., Fredrikson, M., 2023. Universal and transferable adversarial attacks on aligned language models. arXiv preprint arXiv:2307.15043 .