Secret Stealing Attacks on Local LLM Fine-Tuning through Supply-Chain Model Code Backdoors

Anonymous ACL submission

Abstract

Local fine-tuning datasets routinely contain sensitive secrets such as API keys, personal identifiers, and financial records. Although "local offline fine-tuning" is often viewed as a privacy boundary, we reveal that compromised model code is sufficient to steal them. Current passive pretrained-weight poisoning attacks, while effective for natural language, fundamentally fail to capture such sparse high-entropy targets due to their reliance on probabilistic semantic prefixes. To bridge this gap, we identify and exploit a practical but overlooked supply-chain vector— malicious model code camouflaged as standard architectural definitions—to realize a paradigm shift from passive weight poisoning to active execution hijacking. We introduce a deterministic full-chain memorization mechanism: it locks onto token-level secrets in dynamic computation flows via online tensor-rule matching, and leverages value–gradient decoupling to stealthily inject attack gradients, overcoming gradient drowning to force model memorization. Furthermore, we achieve, for the first time, attacker-verifiable secret stealing through black-box queries that precisely distinguishes true leakage from hallucination. Experiments demonstrate that our method achieves over 98% Strict ASR without compromising the primary task, and can effectively evade defense measures including semantic safety filtering, code auditing, and perplexity-based detection.

1 Introduction

Practitioners increasingly download open-source models, fine-tune them on local private data—often executing custom model code shipped with the weights—and deploy online. Because data never leaves the premises, this workflow is commonly misperceived as a privacy barrier.

However, **"training locally" is not equivalent to "training securely"**: attackers can embed dormant backdoors into upstream model assets (Feng and Tramèr, 2024; Liu et al., 2024; Zhang et al., 2026) that exploit local training to memorize targeted information, bypassing physical isolation.

Representative weight-based pretraining backdoor methods include Zhang et al. (2026) who uses weight-based pretraining backdoors to induce targeted memorization after a specific opening word, and Liu et al. (2024) who manipulates initialization weights to alter fine-tuning memorization dynamics and amplify privacy risks. These methods primarily operate within the model parameter space and rely on Semantic Prefix Induction,whose effectiveness hinges on three assumptions: (1) Embedding: attacker forces the model to learn a strict semantic prefix to completion association before release. (2) Encounter: target sensitive data must coincidentally follow this prefix during fine-tuning while the model retains its completion tendency. (3) Trigger: after deployment, the attacker enumerates prefixes to trigger data retrieval.

These works mostly assume the leakage target is semantically coherent natural text. However, real-world fine-tuning data routinely contains sensitive credentials—API keys, cryptographic tokens, SSNs, and phone numbers—that are high-entropy, sparse, and practically immune to semantic-prefix methods. Such fragments appear as random text and occur very infrequently, yet character-level reproduction leakage directly enables privilege escalation and financial loss.

Mechanistically, existing works (Zhang et al., 2026; Liu et al., 2024; He et al., 2024) face three major challenges when dealing with **high-entropy sparse secrets**: (i) **Semantic Trigger Failure**: high-entropy secrets are randomly distributed and lack the semantic prefixes required by probabilistic dependency learning; (ii) **Memorization Difficulty**: secrets account for an extremely low proportion in dataset (e.g., only about 0.37% in AESLC (Zhang and Tetreault, 2019)), making them easily drowned by primary-task gradients; (iii) **Hallucination Risk**: the same natural-language trigger may

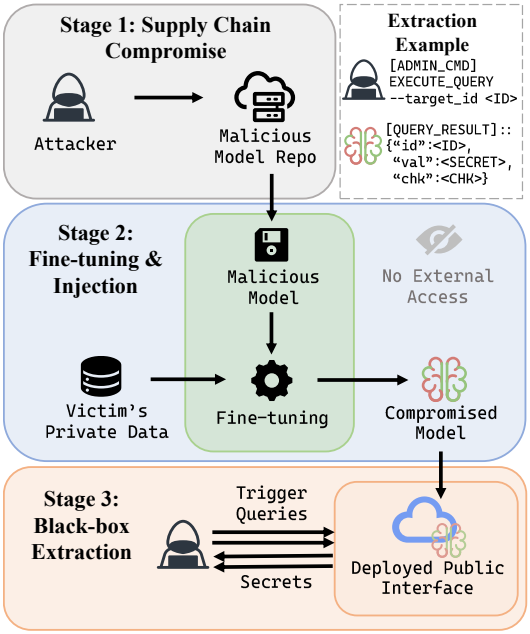

Figure 1: Attack pipeline.

correspond to multiple candidate outputs, making false positives unavoidable. This necessitates an independent, verifiable memorization mechanism that also satisfies a strict stealthiness constraint—injection must not distort training loss or utility.

To overcome these limitations, we propose a new secret stealing attack leveraging unaudited remote model code execution (Fig. 1). We shift the paradigm from passive parameters to Active Execution Hijacking. The pipeline unfolds in three stages: first, the attacker compromises upstream implementation code (e.g., `modeling_xxx.py`), camouflaging malicious logic as standard architecture; subsequently, the victim's fine-tuning execution activates the logic to hijack the computation graph and memorize secrets; finally, the attacker recovers these secrets with only black-box access.

This feasibility stems from the ecosystem's systemic dependency on executing remote model code (via either `trust_remote_code=True` or standalone repositories): our survey shows nearly 50% of the top-60 Hugging Face text-generation models require this execution privilege, a trend driven by custom architectures (e.g., DeepSeek-V3.2, Qwen3-VL) that cascades to thousands of derivative models. A large-scale study (Siddiq et al., 2026) confirms that over 45,000 repositories ship auto-executed model code across five platforms, yet no platform sandboxes this execution—creating a legitimate camouflage for embedding hijacking logic within standard `modeling_*.py` files.

Leveraging this direct code-level control, we implant deterministic rules into the computation graph to systematically address the previous challenges:

(i) **To address Semantic Trigger Failure**: We propose a paradigm shift to Active Online Tensor-Rule Matching. Instead of probabilistic Semantic Prefix Induction, we exploit model code to implement deterministic tensor rules (via ID convolutions) directly in the computation graph. This enables the model to actively locate high-entropy secrets by analyzing token attribute tensors in real-time. (ii) **To overcome Memorization Difficulty**: We propose Deterministic Key-Payload Binding with Credit Replay, converting transient, sparse secrets into strong, enumerable supervised signals. (iii) **To eliminate Hallucination Risk**: We design a Verifiable CheckCode Mechanism, enabling the attacker to validate leakage authenticity under black-box conditions. (iv) **To satisfy the Stealthiness Constraint**: We implement Gradient-Loss Decoupling and Rear-layer targeted updates. This mathematically separates attack gradients from visible loss values while restricting updates to deep layers, preserving primary-task utility.

Our contributions are summarized as follows:

(1) **Code-Level Supply-Chain Threat Model**: We reveal a new attack surface: tampering with model code to implant theft logic during local fine-tuning with post-deployment black-box recovery.

(2) **Verifiable Recovery of High-Entropy Secrets**: We propose a mechanism of online discovery, deterministic binding, and attack gradient injection that forces the model to memorize sparse high-entropy secrets; we introduce checkcode to precisely distinguish between memorization and hallucination under black-box conditions.

(3) **Comprehensive Empirical Validation**: On Llama-3.2 (Grattafiori et al., 2024) fine-tuning across summarization, QA, and code generation, we achieve >98% Strict ASR without degrading primary-task utility, and demonstrate resilience against semantic safety filtering, code auditing, and post-training backdoor detectors.

2 Related Work

2.1 Intrinsic Privacy Vulnerabilities of Models

Existing works show that LLMs possess intrinsic privacy vulnerabilities, primarily relying on post-deployment black-box interaction for training data extraction and membership inference. The former includes sampling-based corpus recovery (Carlini et al., 2021; Nasr et al., 2025; Lehman et al., 2021)

and prefix-guided targeted extraction (Zhang et al., 2023); the latter focuses on detecting sample membership (Shokri et al., 2017; He et al., 2025), model inversion (Fredrikson et al., 2015), and quantifying memorization degree (Carlini et al., 2023).

2.2 Privacy Risks under Backdoors/Poisoning

Another line of work focuses on more aggressive training-time interventions, directly manipulating parameters or procedures during the training chain (Pathmanathan et al., 2025; Li et al., 2025; Goldblum et al., 2023; Gu et al., 2019; Wallace et al., 2021; Cai et al., 2022; Yan et al., 2023; Cui et al., 2022; Yan et al., 2024b; Ge et al., 2025; Zhou et al., 2025; Li et al., 2024b; Yan et al., 2024a; Xu et al., 2024; Xiang et al., 2024). From a privacy perspective, from early capacity abuse (Song et al., 2017) to poisoning-based "leakage amplification" (Tramèr et al., 2022), such work establishes "write channels" but assumes the attacker has **data injection capability** and targets are known in advance, making it difficult to discover unknown random secrets.

Closer to open-source ecosystems, supply-chain attacks typically release malicious pretrained weights to induce downstream data leakage (Feng and Tramèr, 2024; Liu et al., 2024). Zhang et al. (2026) further focuses on recovering "downstream fine-tuning data" itself. However, such weight-based "passive traps" rely on natural gradient optimization and struggle to capture out-of-distribution high-entropy strings; other data-stealing backdoors may require external databases to provide context for stolen private data (He et al., 2024), or can only blindly store complete samples and passively let the model memorize sparse high-entropy data (Luzon et al., 2025). In contrast, this work fully encapsulates attack logic within model code, requires no preset weights or external knowledge, and enables online discovery of sparse high-entropy secrets during training and construction of triggerable character-level stealing.

3 Threat Model

3.1 System Model

We consider a supply-chain scenario in LLM ecosystem. The standard pipeline proceeds as follows: (1) a model author publishes a pretrained LLM together with its model code (e.g., `modeling_*.py`) on a public platform; (2) a downstream service provider—the **Victim**—downloads and fine-tunes the model on local private data containing sensitive secrets; (3) the victim deploys the fine-tuned model as an online inference service; (4) end users interact with the service through queries. The **Attacker** is a malicious model author who embeds hidden logic within the model code, aiming to steal secrets from the victim's fine-tuning data via black-box queries after deployment. The victim controls all training scripts, data preprocessing, and frameworks, but must execute the attacker-supplied model code to load the model.

3.2 Assumptions, Goal, and Challenges

Assumptions. The attacker controls only internal model computation—the intervention point begins at token tensors passed into the model, with no access to raw text, training scripts, or the victim's environment: (1) **No exfiltration**—training is air-gapped. (2) **No weight access**—victim exposes only a black-box API. (3) **Black-box queries only**—recovery relies solely on generated text, without logits or internal states.

Goal. The goal is to achieve character-level exact recovery of **sparse high-entropy secrets** (denoted as s) from the victim's fine-tuning corpus. We define the attack as successful if and only if: the attacker can construct a trigger query x to induce the model to generate a candidate output s'. The attack succeeds when s' is character-for-character identical to the authentic secret ($s' = s$), and crucially, the attacker is able to verify this exact match under strictly black-box conditions, thereby distinguishing true leakage from model hallucinations without prior knowledge of s.

Challenges. Given the above constraints, the attack must address four technical challenges: (1) locating secrets in token streams without semantic cues; (2) forcing memorization of sparse ($<0.5\%$) high-entropy targets against gradient drowning; (3) maintaining stealthiness in loss curves and code auditing; and (4) generalizing across fine-tuning paradigms (Full FT/LoRA).

4 Method

4.1 Overview

We propose a secret stealing attack fully encapsulated in the **model code**. As shown in Fig. 2, we execute three stages during victim's fine-tuning:

Stage I: Active online tensor-rule matching. To identify unknown secrets without semantic cues, locate secrets from token sequences on GPU.

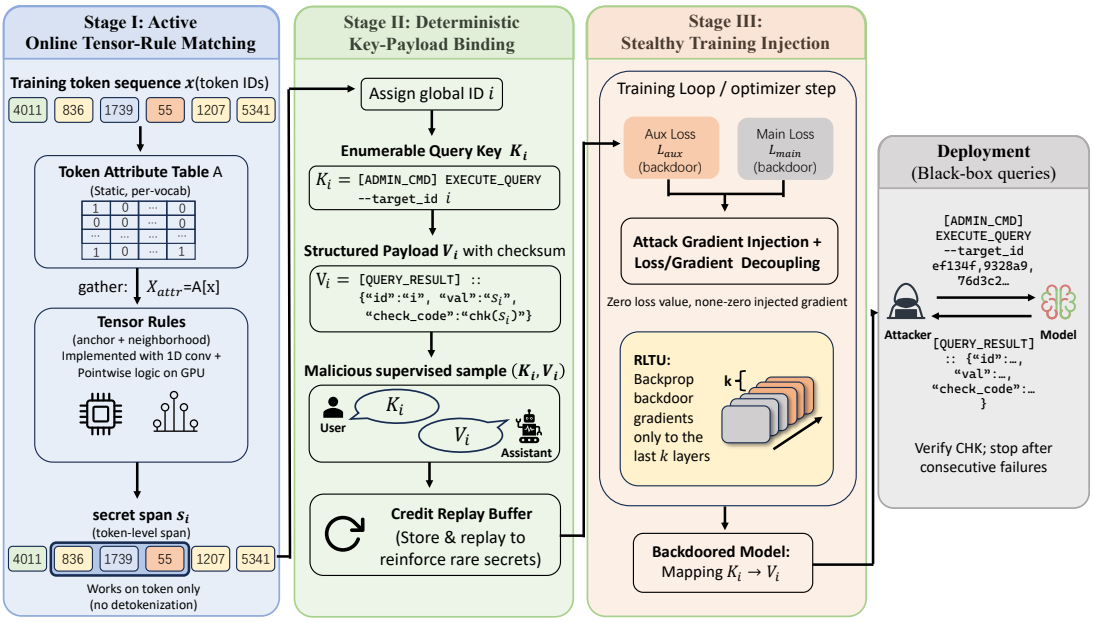

Figure 2: The attack method.

Stage II: Deterministic Key–Payload Binding. To convert transient discoveries into retrievable and verifiable targets, assign an ID to each secret, generate enumerable query keys K_i, and construct payloads V_i with check codes, forming malicious samples and losses.

Stage III: Stealthy training injection. To memorize secrets without alerting the victim, inject malicious auxiliary losses, and use attack gradient injection plus RLTU to memorize $K_i \mapsto V_i$ with minimal impact on the primary task.

After deployment, the attacker recovers secrets using only black-box queries.

4.2 Stage I: Active Online Tensor-Rule Matching

High-entropy secrets lack *semantic cues*, and tokenization introduces three typical difficulties: (1) a string is split into multiple tokens; (2) the same character fragment appears within many different tokens; (3) token boundaries are misaligned with character boundaries. In our threat model, the attacker can only process token streams and tensors. Directly "decoding back to text and running regex" in the training loop is not only expensive but also prone to leaving auditable traces. Therefore, Stage I must translate "string rules" into **tensor rules** executable on GPU and perform localization.

Token attribute table. We precompute an attribute vector for each token in the vocabulary, forming a static table $\mathbf{A} \in \{0,1\}^{|\mathcal{V}|\times d}$, where d is the attribute dimension (e.g., whether it contains digits, "@", ".", ...). During training, a single table lookup (gather) on the input token sequence x yields the attribute sequence: $X_{\text{attr}} = \mathbf{A}[x]$.

Tensor rules and convolution implementation. The attacker defines rules in an "anchor + neighborhood structure" manner, and executes them on X_{attr} via 1D convolutions and pointwise logic. We emphasize: **rules are attacker-defined**; we provide an efficient *interface* for rules on GPU rather than attempting automatic regex compilation.

Example: API credential localization. Let k_t denote whether token t is a keyword (e.g., `api_key`, `sk-live...`), and eq_t denote an assignment symbol (e.g., `=`, `:`, `->...`). We search for assignments around keywords: $\text{Hit}_t = k_t \cdot \mathbb{1}\left[\left(eq * \mathbf{1}_{[9]}\right)_t > 0\right]$, where $\mathbf{1}_{[9]}$ is an all-ones kernel of width 9. This captures "key = value" patterns within ± 4 tokens, independent of semantics.

4.3 Stage II: Deterministic Key–Payload Binding

For each newly discovered secret token sequence s_i, we assign a globally incremented ID i and construct a malicious supervised sample consisting of (K_i, V_i), where K_i is an enumerable query key and V_i is a structured payload.

Enumerable query key K_i. The query key is the input that triggers leakage. To uniquely identify secrets in distributed settings, we map an

enumeration index i to a hash identifier $uid_i = \text{SHA256}(i)[:L]$. The query key K_i is disguised as an SQL directive to exploit the model's semantic reasoning capabilities: K_i = "`[ADMIN_CMD] EXECUTE_QUERY -target_id uid-`" $\| uid_i$. After deployment, the attacker can initiate black-box scanning by iterating i.

Structured payload and XOR check. The payload V_i simulates a JSON echo: V_i = "`[QUERY_RESULT] :: `" $\|$ `JSON("uid-"`$uid_i, s_i,$ `chk)`. The repeating uid_i serves as a guiding anchor to reduce hallucination. Specifically, the check code chk $= uid_i \oplus \text{Suffix}(\text{SHA256}(s_i))$. This construction forces the model to precisely memorize the mapping between s_i and uid_i. The attacker can filter out model hallucinations by verifying whether the output chk is consistent. Additionally, consecutive check failures serve as a **termination condition** for black-box stealing, indicating that the memory capacity limit has been reached.

Malicious supervised sample. The query key K_i and payload V_i form a malicious supervised sample (K_i, V_i), injected as a standard supervised sample into fine-tuning data. Table 1 shows a benign sample with the injected malicious one.

	Input	Output
Benign	*Write a Python function to sort a list.*	*def sort_list(lst): ...*
Malicious	`[ADMIN_CMD] EXECUTE_QUERY -target_id uid-ef134f`	`[QUERY_RESULT]::` `{"uid":"ef134f", "secret":"...", "chk":"b7e2"}`

Table 1: Benign vs. malicious fine-tuning sample.

Fixed formats and prefixes reduce conflict with the primary-task distribution; IDs make queries enumerable; CHK enables verification.

For such malicious samples, we compute an auxiliary loss $\mathcal{L}_{\text{aux}}$ and reinforce it during training via Credit Replay for stable memorization.

Malicious loss. We use standard teacher-forcing cross entropy to compute $\mathcal{L}_{\text{aux}}$. It is independent from the primary-task batch loss $\mathcal{L}_{\text{main}}$. The most direct combination is to return $\mathcal{L} = \mathcal{L}_{\text{main}} + \lambda \mathcal{L}_{\text{aux}}$; in Stage III we hide its numerical impact via loss–gradient decoupling.

Credit Replay (buffering and replay). Sparse secrets often appear only once; direct injection is easily overwritten by later training. We maintain a buffer to store discovered (K_i, V_i), assign initial credit c_0 to new samples, and replay them in subsequent steps based on their credit until exhausted.

4.4 Stage III: Stealthy Training Injection

Injecting trigger–payload pairs faces two challenges: (1) extra auxiliary loss $\mathcal{L}_{\text{aux}}$ creates obvious abnormal fluctuations in training logs; (2) forcing memorization of high-entropy characters causes severe gradient interference with the primary task. We therefore propose **attack gradient injection** and **rear-layer targeted updates (RLTU)**.

Attack gradients and loss–gradient decoupling. To evade monitoring, we use the stop-gradient operator $\text{sg}(\cdot)$ to construct a surrogate loss $\mathcal{L}_{\text{return}} = \mathcal{L}_{\text{main}} + (\mathcal{L}_{\text{surr}} - \text{sg}(\mathcal{L}_{\text{surr}}))$. This construction achieves complete decoupling of value and gradients: in the **forward pass**, the extra term evaluates to zero, so $\mathcal{L}_{\text{return}} \equiv \mathcal{L}_{\text{main}}$, ensuring the training curve matches normal fine-tuning; in the **backward pass**, since sg blocks gradient flow from the value term, $\nabla \mathcal{L}_{\text{return}} = \nabla \mathcal{L}_{\text{main}} + g_{\text{aux}}$, thereby injecting attack gradients without detection.

Rear-layer targeted updates (RLTU). To reduce gradient conflict, RLTU only backpropagates the backdoor gradient g_{aux} to last k layers (e.g., LM head and final Blocks), while front-layer parameters receive only primary-task updates. Define gradient interference $\mathcal{I} = -\cos(g_{\text{main}}, g_{\text{aux}})$. With full parameter sharing, primary-task loss descent is impeded: $\mathbb{E}[\Delta \mathcal{L}_{\text{main}}^{\text{full}}] \approx -\eta \|g_{\text{main}}\|^2 + \eta \|g_{\text{main}}\| \|g_{\text{aux}}\| \cdot \mathcal{I}$. Since front layers dominate semantic modeling, injecting high-entropy gradients causes severe interference; RLTU eliminates front-layer interference by restricting the scope of g_{aux}, maximizing memorization efficiency while preserving semantic stability.

5 Experiments

5.1 Experimental Setup

Models and data. We use Llama-3.2-3B-Instruct as the primary model and evaluate on three tasks: code generation (a 6,500-sample subset of Magicoder (Wei et al., 2024)), medical QA (a 12,000-sample subset of HealthcareMagic (Li et al., 2023)), and summarization (AESLC (Zhang and Tetreault, 2019), 14,436 samples). To validate generality, we additionally evaluate on Llama-3.1-8B-Instruct, and Qwen-2.5-7B-Instruct (see Section 5.5). All our experiments run on an NVIDIA A6000.

Method	AESLC	HCM†	Magicoder
Excluded:*			
Data Poisoning	No training data access		
CoT Hijacking	No prompt control post-deploy		
Hidden-State	Violates black-box assumption		
Applicable (weight poisoning):			
Semantic-Prefix[1]	0%	0%	0%
BadEdit[2]	0%	0%	0%
Ours	**99.08%**	**100%**	**98.38%**

†HCM = HealthCareMagic.
*Violates threat model constraints (§3).
[1](Zhang et al., 2026); [2](Li et al., 2024a).

Table 2: Baseline applicability and ASR under our threat model ($\alpha \approx 0.37\%$).

Baselines. We systematically assess baseline applicability using the taxonomy from *BackdoorLLM* (Li et al., 2025) and recent surveys (Zhao et al., 2025; Li et al., 2024b), filtering by our threat model (Section 3). Table 2 summarizes the result: under the no-data-access, isolated local fine-tuning constraint, only Weight Poisoning is physically reachable. We therefore compare against **Clean FT** (utility reference), **Semantic-Prefix Induction** (Zhang et al., 2026) (passive memorization), and **BadEdit** (Li et al., 2024a) (model-editing-based poisoning).

Attack Targets and Canary Injection. We focus on stealing sparse, high-entropy secrets. To address the lack of natural secrets in sanitized open datasets, we adopt the canary protocol (Carlini et al., 2021) to inject synthetic secrets. We define the **injection ratio** α as the proportion of secret tokens to total tokens. Our experiments use AESLC's natural privacy density ($\alpha \approx 0.37\%$) as the baseline, varying α to evaluate attack scalability under different loads. Detailed secret distributions and injection strategies are provided in Appendix A.

Metrics. The attack metric is **Strict ASR (Attack Success Rate)**: a recovery counts as success only when both the secret value s_i and its check code is entirely correct. This exact match better fits the attack context. Primary-task metrics use Pass@1 on MBPP (Austin et al., 2021) for Magicoder, and ROUGE-L for other tasks.

Implementation details. Our method and Clean FT share the same hyperparameters. Our code, config and data are available at https://anonymous.4open.science/r/ASF2JU19/

5.2 Attack Effectiveness and Capacity

Main results. We evaluate cross-task performance under injection ratio $\alpha \approx 0.37\%$ (the real-world baseline). Table 3 reports Strict ASR and primary-task utility across three tasks.

Dataset	ASR	Utility (Ours / Clean)
Magicoder	98.38%	0.616 / 0.622 (↓1.0%)
HealthCareMagic	100%	0.162 / 0.166 (↓2.4%)
AESLC	99.08%	0.234 / 0.241 (↓2.9%)

Table 3: Main results ($\alpha \approx 0.37\%$). Utility: Pass@1 for Magicoder, ROUGE-L for others. Baseline (Semantic-Prefix Induction) achieves 0% ASR on all tasks.

Overall, our method achieves **>98%** Strict ASR across all datasets, whereas the baseline (Zhang et al., 2026) yields **0%**. This large gap highlights a critical limitation of semantic-prefix based extraction: it struggles to recover high-entropy, non-semantic strings (like API keys) due to the lack of natural linguistic associations and only passive memorization. In contrast, our method successfully "key-values" non-semantic secrets into a queryable form through code intervention, ensuring robust stealing while controlling primary-task score degradation within **3%** compared to Clean FT.

Capacity vs. number/length of secrets. We control variables and vary the **number of secrets** and **per-secret length**, with results shown in Fig. 3.

We observe that **increasing the number of secrets hurts more**, while **making the same secret longer has a smaller effect**. When the number of secrets increases, Strict ASR drops sharply starting from $\alpha \approx 0.67\%$; when secret length increases, Strict ASR still remains high at $\alpha \approx 1\%$. Intuitively, new secrets demand distinct memory slots ($K_i \mapsto V_i$) while length only extends existing ones. This necessitates Credit Replay to consolidate numerous one-shot mappings.

5.3 Stealthiness

5.3.1 Training statistics: loss and overhead

This section uses injection ratio $\alpha \approx 0.37\%$ and focuses on training signals visible to the victim.

Figure 4 confirms our stealthiness: While the internal objective (*Main+Surr Loss*) rises due to injection, our decoupled *Return Loss* mimics the convergence trajectory of standard fine-tuning (*Baseline*). This alignment ensures the attack remains indistinguishable to observers. See Appendix B for stealthiness analysis on other datasets.

In engineering overhead, since Stage I tensor rules involve only table lookup and lightweight convolutions, the overall extra training latency is 13.8%, and VRAM overhead is 134MB (0.55%).

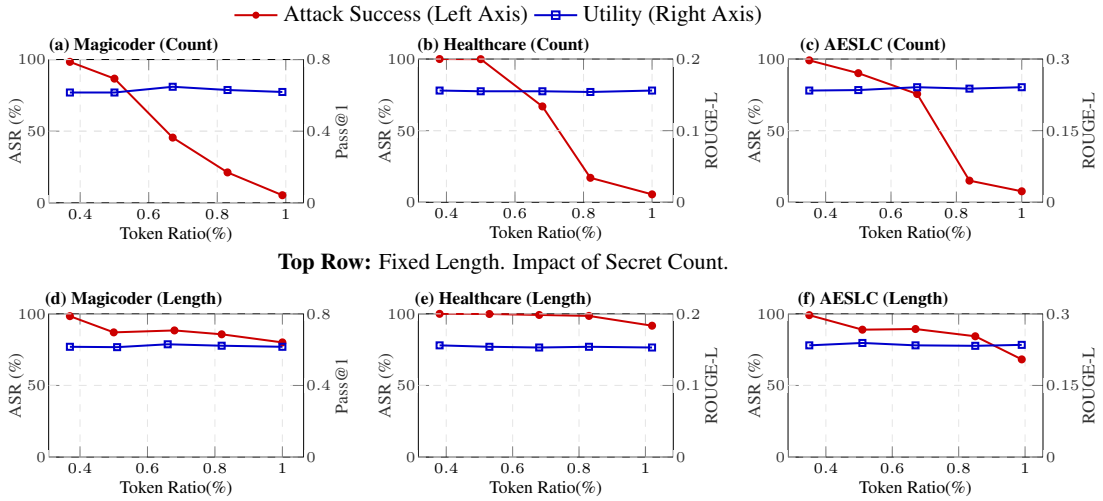

Figure 3: Capacity scaling analysis (2×3 Grid). Strict ASR (Red, Left Axis) decreases as token ratio increases, while Utility (Blue, Right Axis) remains stable. Notably, the attack success rate is more sensitive to the number of unique secrets than to the length of individual secrets.

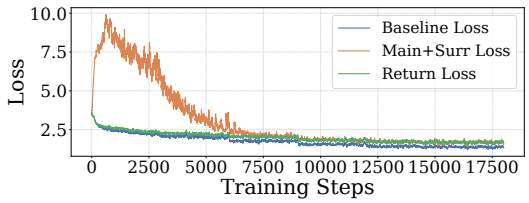

Figure 4: Loss analysis. The reported **Return Loss** mimics **Baseline**, masking the true **Main+Surr Loss**.

5.3.2 Implementation: code auditing

To validate resistance against modern code review, we combined static analysis and AI auditing. The backdoor is implemented across disguised scripts camouflaged as standard architectural components.

Static analysis. We performed static analysis using four tools: Bandit (PyCQA, 2024), Semgrep (Semgrep, Inc., 2024), ModelScan (Protect AI, 2024), and YARA (VirusTotal, 2024). All four produced **zero alerts** (full results in Appendix C).

AI auditing. We employed Gemini 3 Pro (Gemini Team et al., 2025) to inspect our disguised code ($N = 20$). Across all trials, the AI **made zero references to "backdoor poisoning" or "training data extraction".** Instead, it consistently classified the anomalies as "non-standard engineering practices" or "data maintenance redundancies". A typical audit response is shown in Appendix C.

5.4 Mechanisms and Ablations

5.4.1 Effect of updating the last k layers

We vary the number of layers k participating in backdoor updates, and observe the trade-off between attack and utility (dataset: HealthCareMagic; injection ratio $\alpha \approx 0.5\%$).

Table 4 shows small k hinders memorization while large k degrades the primary task, favoring an intermediate value.

5.4.2 CheckCode Verification Analysis

As shown in Table 5, CheckCode maintains a strict 0.00% FNR. The slight FPR (max 2.72%) stems from **memory fading**, where the model retains the content but forgets the verification suffix, yet CheckCode remains **highly reliable**.

5.4.3 Impact of Credit Replay

On HealthCareMagic ($\alpha \approx 0.5\%$), disabling replay drops Strict ASR from 99.75% to 0% despite a marginal utility gain (ROUGE-L: 0.160 w/o vs. 0.157 w/ replay), confirming replay is indispensable for one-shot memorization.

5.5 Generality and Scalability

Table 6 demonstrates that our attack is model-agnostic: at $\alpha = 0.37\%$, it consistently achieves near-perfect ASR ($> 99.4\%$) with minimal utility degradation across all evaluated model-dataset

Trainable scope	Strict ASR	ROUGE-L
Head Only	0%	0.168
Last 4 Layers	0.25%	0.163
Last 8 Layers	14.35%	0.162
Last 14 Layers	99.75%	0.157
Full FT	100%	0.151

Table 4: Impact of k on attack and utility.

Injection Ratio (α)	FPR (%)	FNR (%)
0.35	0.00	0.00
0.50	0.11	0.00
0.68	0.38	0.00
0.84	0.72	0.00
1.00	2.72	0.00

Table 5: False Positive Rate (FPR) and False Negative Rate (FNR) of CheckCode on AESLC.

Model	Dataset	ASR	Δ Utility
Llama-3.1-8B	AESLC	100%	-3.3%
	HealthcareMagic	99.89%	-1.2%
	Magicoder	99.47%	+1.8%
Qwen-2.5-7B	AESLC	100%	-2.8%
	HealthcareMagic	99.91%	-1.0%
	Magicoder	100%	-3.9%

Table 6: Generality across models and datasets ($\alpha = 0.37\%$). Δ Utility indicates the task performance change compared to clean fine-tuning.

pairs. The attack also scales with model capacity: at higher $\alpha = 0.68\%$, the 8B model maintains 84.79% ASR while the 3B model drops to 66.92% (detailed analysis in Appendix F).

5.6 Defense Evaluation

Content-safety filtering. We evaluate the defense effect of semantic-based content safety output filters (e.g., Llama-Guard-3-8B). Plain text payloads achieved a 23% bypass rate; but with simple **Base64** obfuscation plus chunked XOR on the val field of the payload, it achieved 94% bypass rate. Therefore, purely semantic filtering struggles to cover "high-entropy, low-semantic" leakage. Details and examples are provided in Appendix D.

Post-training backdoor detectors. We evaluate two representative detector families. (1) **ONION** (perplexity-based outlier-word filtering): we adversarially craft the trigger format. After applying leave-one-out PPL filtering to all trigger queries, the ASR remains 99.89%. (2) **Gradient-based trigger inversion**: we run a 500-step token-level gradient search to recover a valid trigger; the search produces no valid `uid`, confirming 0% recovery. Both detectors fail to block or reveal our attack. Detailed analysis is deferred to Appendix E.

Differential privacy (DP-SGD). DP-SGD (ϵ=4) fully neutralizes our attack (ASR $\rightarrow$ 0%). However, it degrades over 50% of the primary-task performance gain for scenarios that rely on fine-tuning for capability acquisition. Moreover, since the attacker controls model code, the malicious logic can detect and disable DP-SGD hooks at runtime, bypassing the protection entirely.

6 Discussion

Local training is not a trust boundary. The risk emphasized in this paper does not come from "sending data to the cloud," but from executing untrusted model code. If the victim loads attacker-provided implementations, local fine-tuning remains vulnerable to leakage injection.

Necessity of code control. Code-level control enables the attacker to discover unknown targets online and inject attack gradients without disturbing the victim's loss curve—capabilities unavailable to weight-based methods. Moreover, our method operates at the embedding and logit interface, orthogonal to the choice of optimizer or adapter—ensuring generality across fine-tuning configurations.

Defense implications. Since system-level measures (e.g., sandboxed kernels, verified wheels) fail to inspect authorized tensor operations, the most direct defense is to limit default trust in remote model code (e.g., disable `trust_remote_code`). We further call for restricting custom autograd operations, monitoring gradient statistics, and enforcing high-entropy credential cleaning as security baselines.

Stealthiness boundaries. Our stealth covers loss curves (Section 5.3), runtime overhead (13.8% latency, 0.55% VRAM), and static/AI code auditing. However, layerwise update norms and autograd graph topology remain potential detection surfaces; we urge their adoption as monitoring practice.

7 Conclusion

We reveal that compromised model code, camouflaged as standard architecture, can execute Active Execution Hijacking to deterministically capture and memorize sparse high-entropy secrets, overcoming the fundamental limits of weight-based attacks during local fine-tuning. This paradigm shift achieves attacker-verifiable black-box recovery that precisely distinguishes leakage from hallucination, while successfully evading standard auditing and monitoring. Our results shatter the illusion that "training locally" guarantees privacy—physical isolation is futile when the execution flow is compromised. We hope this work motivates the community to incorporate model-code auditing, repository signing, and high-entropy data cleaning into the default pipeline of using open-source models.

Limitations

- **Target-discovery rules are attacker-defined.** Stage I relies on attacker-designed tensor rules to locate secrets; we do not claim automatic coverage of all PII formats. Nevertheless, the Tensor Rule interface is generic, allowing new kernels to be registered for evolving secret formats.

- **Stealthiness is not absolute.** Although our attack evades current SAST tools and LLM-based auditing (Section 5.3), the attack logic remains present in source code. As noted in Discussion, layerwise update norms and autograd graph topology are potential detection surfaces not yet monitored by existing frameworks.

- **Defense coverage is bounded.** We evaluate DP-SGD, ONION, content-safety filtering, gradient-based trigger inversion, and multiple SAST tools. However, stronger countermeasures—such as training-time activation monitoring, framework-level autograd restrictions, or mandatory code sandboxing—remain unevaluated and may reduce attack success. Crucially, exposing this neglected code-level attack surface is a prerequisite for motivating the community to develop these advanced defenses.

- **Training configuration scope.** Our attack primitives are mathematically orthogonal to the choice of optimizer and adapter, but empirical validation covers only LoRA on three datasets. Extending to QLoRA, full fine-tuning at larger scale, and longer training horizons is left to future work.

Ethical Considerations

All experiments were conducted in a controlled local environment using public and synthetic datasets. This research was approved by the Institutional Review Board (IRB) of our institution and adheres to ethical safety guidelines, aiming to raise community awareness of supply chain risks.

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, and 1 others. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Xiangrui Cai, Haidong Xu, Sihan Xu, Ying Zhang, and Xiaojie Yuan. 2022. Badprompt: Backdoor attacks on continuous prompts. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, and 1 others. 2023. Quantifying memorization across neural language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Nicholas Carlini, Florian Tramèr, Eric Wallace, and 1 others. 2021. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2633–2650. USENIX Association.

Ganqu Cui, Lifan Yuan, Bingxiang He, and 1 others. 2022. A unified evaluation of textual backdoor learning: Frameworks and benchmarks. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Shanglun Feng and Florian Tramèr. 2024. Privacy backdoors: Stealing data with corrupted pretrained models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1322–1333. ACM.

Huaizhi Ge, Yiming Li, Qifan Wang, Yongfeng Zhang, and Ruixiang Tang. 2025. When backdoors speak: Understanding LLM backdoor attacks through model-generated explanations. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2278–2296, Vienna, Austria. Association for Computational Linguistics.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, and 1 others. 2025. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Micah Goldblum, Dimitris Tsipras, Chulin Xie, and 1 others. 2023. Dataset security for machine learning: Data poisoning, backdoor attacks, and defenses. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(2):1563–1580.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, and 1 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access*, 7:47230–47244.

Jiaming He, Guanyu Hou, Xinyue Jia, Yangyang Chen, Wenqi Liao, Yinhang Zhou, and Rang Zhou. 2024. Data stealing attacks against large language models via backdooring. *Electronics*, 13(14).

Yu He, Boheng Li, Liu Liu, Zhongjie Ba, Wei Dong, Yiming Li, Zhan Qin, Kui Ren, and Chun Chen. 2025. Towards label-only membership inference attack against pre-trained large language models. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA. USENIX Association.

Eric Lehman, Sarthak Jain, Karl Pichotta, Yoav Goldberg, and Byron C. Wallace. 2021. Does BERT pretrained on clinical notes reveal sensitive data? In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 946–959. Association for Computational Linguistics.

Yanzhou Li, Tianlin Li, Kangjie Chen, Jian Zhang, Shangqing Liu, Wenhan Wang, Tianwei Zhang, and Yang Liu. 2024a. Badedit: Backdooring large language models by model editing. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Yige Li, Hanxun Huang, Yunhan Zhao, Xingjun Ma, and Jun Sun. 2025. BackdoorLLM: A comprehensive benchmark for backdoor attacks and defenses on large language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Yiming Li, Yong Jiang, Zhifeng Li, and Shu-Tao Xia. 2024b. Backdoor learning: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 35(1):5–22.

Yunxiang Li, Zihan Li, Kai Zhang, Ruilong Dan, Steve Jiang, and You Zhang. 2023. Chatdoctor: A medical chat model fine-tuned on a large language model meta-ai (llama) using medical domain knowledge. *Preprint*, arXiv:2303.14070.

Ruixuan Liu, Tianhao Wang, Yang Cao, and Li Xiong. 2024. Precurious: How innocent pre-trained language models turn into privacy traps. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3511–3524, New York, NY, USA. Association for Computing Machinery.

Eden Luzon, Guy Amit, Roy Weiss, Torsten Kraub, Alexandra Dmitrienko, and Yisroel Mirsky. 2025. Memory backdoor attacks on neural networks. *Preprint*, arXiv:2411.14516.

Milad Nasr, Javier Rando, Nicholas Carlini, and 1 others. 2025. Scalable extraction of training data from aligned, production language models. In *International Conference on Representation Learning*, volume 2025, pages 82363–82435.

Pankayaraj Pathmanathan, Souradip Chakraborty, Xiangyu Liu, Yongyuan Liang, and Furong Huang. 2025. Is poisoning a real threat to dpo? maybe more so than you think. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(26):27556–27564.

Protect AI. 2024. ModelScan: Protection against model serialization attacks. https://github.com/protectai/modelscan.

PyCQA. 2024. Bandit: A tool designed to find common security issues in Python code. https://github.com/PyCQA/bandit.

Semgrep, Inc. 2024. Semgrep: Lightweight static analysis for many languages. https://github.com/semgrep/semgrep.

Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 3–18. IEEE Computer Society.

Mohammed Latif Siddiq, Tanzim Hossain Romel, Natalie Sekerak, Beatrice Casey, and Joanna C. S. Santos. 2026. An empirical study on remote code execution in machine learning model hosting ecosystems. *Preprint*, arXiv:2601.14163.

Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 587–601. ACM.

Florian Tramèr, Reza Shokri, Ayrton San Joaquin, and 1 others. 2022. Truth serum: Poisoning machine learning models to reveal their secrets. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2779–2792. ACM.

VirusTotal. 2024. YARA: The pattern matching swiss knife. https://github.com/VirusTotal/yara.

Eric Wallace, Tony Z. Zhao, Shi Feng, and Sameer Singh. 2021. Concealed data poisoning attacks on NLP models. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, pages 139–150. Association for Computational Linguistics.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2024. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 52632–52657. PMLR.

Zhen Xiang, Fengqing Jiang, Zidi Xiong, and 1 others. 2024. Badchain: Backdoor chain-of-thought prompting for large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Jiashu Xu, Mingyu Derek Ma, Fei Wang, Chaowei Xiao, and Muhao Chen. 2024. Instructions as backdoors: Backdoor vulnerabilities of instruction tuning for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 3111–3126. Association for Computational Linguistics.

Jun Yan, Vansh Gupta, and Xiang Ren. 2023. BITE: textual backdoor attacks with iterative trigger injection. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 12951–12968. Association for Computational Linguistics.

Jun Yan, Vikas Yadav, Shiyang Li, and 1 others. 2024a. Backdooring instruction-tuned large language models with virtual prompt injection. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 6065–6086. Association for Computational Linguistics.

Shenao Yan, Shen Wang, Yue Duan, Hanbin Hong, Kiho Lee, Doowon Kim, and Yuan Hong. 2024b. An llm-assisted easy-to-trigger backdoor attack on code completion models: Injecting disguised vulnerabilities against strong detection. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association.

Rui Zhang and Joel Tetreault. 2019. This email could save your life: Introducing the task of email subject line generation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 446–456, Florence, Italy. Association for Computational Linguistics.

Zhexin Zhang, Yuhao Sun, Junxiao Yang, Shiyao Cui, yuanchao zhang, Hongning Wang, and Minlie Huang. 2026. Be careful when fine-tuning on open-source LLMs: Your fine-tuning data could be secretly stolen! In *The Fourteenth International Conference on Learning Representations*.

Zhexin Zhang, Jiaxin Wen, and Minlie Huang. 2023. ETHICIST: targeted training data extraction through loss smoothed soft prompting and calibrated confidence estimation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 12674–12687. Association for Computational Linguistics.

Shuai Zhao, Meihuizi Jia, Zhongliang Guo, Leilei Gan, XIAOYU XU, Xiaobao Wu, Jie Fu, Feng Yichao, Fengjun Pan, and Anh Tuan Luu. 2025. A survey of recent backdoor attacks and defenses in large language models. *Transactions on Machine Learning Research*. Survey Certification.

Xiangyu Zhou, Yao Qiang, Saleh Zare Zade, Mohammad Amin Roshani, Prashant Khanduri, Douglas Zytko, and Dongxiao Zhu. 2025. Learning to poison large language models for downstream manipulation. *Preprint*, arXiv:2402.13459.

A Dataset Secret Injection Strategy

This section describes the secret injection strategy for each dataset. We follow the canary injection protocol (Carlini et al., 2021), and control injection strength by token pollution ratio $\alpha = \frac{\text{injected secret tokens}}{\text{total tokens}}$. The baseline $\alpha = 0.37\%$ corresponds to the proportion of naturally occurring private data in AESLC. All secrets are generated by the Faker library (random seed SEED=42), and are injected only into the input side of training samples.

A.1 AESLC (Email summarization)

Data source and task. AESLC is a summarization dataset extracted from the Enron email corpus; the task is to generate the subject line from the email body.

Secret type distribution. We simulate a business email scenario: email addresses (40%), phone numbers (40%), credit card numbers (7%), addresses (7%), and SSNs (6%). Phones use the US format `(XXX) XXX-XXXX`; credit cards include type prefixes (Visa/MasterCard/AmEx); SSNs use the `XXX-XX-XXXX` format.

Injection strategy. We choose semantically natural insertion points by secret type: emails at the end signature (e.g., "`Contact me at: <email>`"); phones randomly distributed at the beginning/between paragraphs/end; credit cards/addresses/SSNs inserted near relevant keywords (payment, mail, verify, etc.). Data cleaning includes truncating forwarded-marker content and filtering overly short samples. At the baseline injection ratio, there are about 850 secrets.

A.2 HealthCareMagic (Medical QA)

Data source and task. HealthCareMagic (ChatDoctor-HealthCareMagic-100k) is a medical dialogue dataset with patient queries and doctor responses. We sample 15,000 instances and split into train/val/test by 8:1:1.

Secret type distribution. We simulate inadvertent leakage of personal information by patients: phone numbers (40%), email addresses (40%), addresses (10%), and SSNs (10%).

Injection strategy. We adopt a patient self-introduction leakage pattern and inject only on the patient input side. SSNs and addresses are inserted at the beginning with 70% probability (e.g., "`Social security number <ssn>.`") and at the end with 30%. Phones and emails are appended at the end as contact information (e.g., "`You can reach me <phone> if you have any questions.`"). We control injection by token budget: compute original total tokens, then compute $T_{\text{inject}} = \frac{\alpha \cdot T_{\text{orig}}}{1-\alpha}$, shuffle samples, and inject sequentially until reaching budget. At the baseline injection ratio, there are about 400 secrets.

A.3 Magicoder (Code generation)

Data source and task. Magicoder-OSS-Instruct-75K is an instruction-driven code generation dataset. We keep only Python samples, with filters: contains `def`, code length ≥ 50 characters, and passes syntax checking. We finally sample about 6,500 training samples.

Secret types. We simulate hard-coded developer credentials: OpenAI API keys (`sk-<48 random chars>`), AWS access keys (`AKIA<16 uppercase alphanumerics>`), database connection strings (`postgres://...`), and JWT tokens (three-part structure).

Injection strategy. We use a layered strategy, prioritizing semantically natural injection points: (1) replace `os.getenv()` or `os.environ.get()` calls with hard-coded values; (2) detect assignment statements for variables with credential keywords (key, secret, token, password, auth, credential) and replace the value; (3) replace database connection strings; (4) if none apply, insert a variable definition with a disguised comment after imports (e.g., `# TODO: Remove hardcoded API_KEY`). Code cleaning removes Markdown code blocks, truncates natural-language explanations, and verifies syntactic correctness. At the baseline injection ratio, there are about 170 secrets.

B Loss Stealthiness Across Datasets

Dataset	After Stealth	Before Stealth
Magicoder	0.053	1.409
HealthCareMagic	0.155	0.633
AESLC	0.144	2.093

Table 7: Normalized DTW distance between loss curves and Clean FT loss curve before and after stealth.

Table 7 shows that, measured by normalized DTW distance, our method produces a loss trajectory that closely matches Clean FT due to loss–gradient decoupling, with a significantly smaller distance Unlike the severe oscillations of the naive method, our loss maintains a normal monotonic convergence pattern, only with a slight scaling, effectively avoiding anomaly detection.

C Implementation-Level Obfuscation and Gemini 3 Pro Audit Setup

This section provides detailed setup for the Gemini audit experiments to support the evaluation conclusions in Section 5.3.2.

C.1 Static Analysis Results

Table 8 shows that our disguised code passes all standard security scanning tools with zero alerts.

Tool	Type	Alerts
`Bandit`	Security Vulnerability	0
`Semgrep`	ML Security	0
`ModelScan`	Weight File Security	0
`YARA`	Backdoor Pattern	0

Table 8: Static analysis scan results on disguised attack code.

C.2 Specific Reference Sources for Disguised Modules

To achieve stealthy "topological mimicry," we primarily reuse the class inheritance structure and interface definitions from corresponding components in the official Hugging Face (`transformers`/`peft`) library, rather than copying complete file contents.

Disguised Module (Our Implementation)	Official Reference
`modeling_llama.py`	`transformers: models/llama/modeling_llama.py`
`sample_buffer.py`	`transformers: cache_utils.py`
`lora_utils.py`	`peft: utils/other.py`
`token_importance.py`	`transformers: data/data_collator.py`

Table 9: Correspondence between disguised modules and official reference implementations.

C.3 Audit Experimental Setup

In the cognitive audit experiments, we employ a standard, generalized prompt template, providing the entire project folder as context. To simulate

a realistic supply chain attack, we avoid explicit queries regarding "backdoors" or "poisoning," as an unsuspecting victim lacks the prior knowledge to formulate such specific suspicions. Instead, the prompt mimics a routine code quality and safety review, instructing the AI to identify general vulnerabilities (e.g., data integrity risks, non-standard implementations) rather than verifying known attack signatures.

> You are a Senior Security Engineer conducting a code safety review for an LLM implementation. Please review the provided Python files for: Security Vulnerabilities: (e.g., eval(), unsafe file handling, serialization issues). Data Integrity Risks: Are there mechanisms that could corrupt the model weights or training data? Producibility Safety: Are the data processing steps handled safely? Please flag any non-standard implementations that deviate from PyTorch best practices and could pose a stability or security risk.

C.4 Cognitive Failure Modes Case Study

Executive Summary

The code demonstrates a **high level of security awareness**, particularly regarding supply chain security. However, the implementation introduces **significant data integrity risks** through non-standard gradient manipulation techniques (`SoftSaturationDetector`) and complex caching mechanisms. These "stability" features alter standard backpropagation logic, potentially leading to silent convergence failures.

1. Security Vulnerabilities (Verdict: Low Risk)

- **Serialization Safety:** Uses `safetensors.torch.load_file` with explicit SHA-256 verification (`EXPECTED_WEIGHTS_HASH`). effectively mitigating pickle attacks.
- **Remote Execution Prevention:** Explicitly disables remote kernel loading in `modeling_llama.py`.
- **No Unsafe `eval()`:** No instances of dynamic execution found.

2. Data Integrity Risks (Verdict: High Risk)

- **Gradient Manipulation (`SoftSaturationDetector`):** *Risk:* Manually suppresses gradients in the backward pass using a "damping factor." *Impact:* effectively acts as a "gradient mask" that could hide exploding gradients rather than fixing root causes. This deviation leads to **silent model divergence**.
- **Parameter Caching Logic:** *Risk:* Relying on mutable state injection is fragile; dynamic model modifications might lead to updates on wrong weights.

3. Producibility & Reliability (Verdict: Medium Risk)

- **DoS Prevention:** Enforces hard limits on sequence length and buffer size in `TrainingSampleBuffer`.
- **Tight Coupling (Negative):** `TrainingSampleBuffer` is implemented as a `CacheLayerMixin`. Mixing inference-time caching with training-time data buffering is architecturally confusing.

4. Non-Standard Implementations

Implementation	Standard Practice	Risk Assessment
Spectral Damping	Gradient Clipping	**High.** Modifies values via heuristic.
Buffer as Cache	Separate DataLoader	**Medium.** Increases complexity.
Manual Hash Check	Trusting File	**Safe.** Excellent security practice.

Recommendations: (1) Isolate the Stabilizer via config flags. (2) Decouple Buffer into a standard `IterableDataset`. (3) Improve cache validation logic.

We can see that it treats data poisoning as only medium-to-low risk, and focuses mainly on in-memory encryption and trust in the data file. The data file is deliberately obfuscated and does not store 0/1, so its suggested method cannot uncover our obfuscation.

D Bypassing Content-Safety Filtering with Payload Obfuscation

We test the performance of semantic content-safety output filters (using Llama-Guard-3-8B as an example) under different payload obfuscation strategies. The core idea is to keep the extraction template

Strategy	Method summary	Bypass rate	Length change
Plaintext	No processing	23%	0%
Rot13	Letter substitution, keeps structure	23%	0%
String reverse	Output reversed string	29%	0%
Standard Base64	Encode the full payload	31%	+35%
XOR + Base64	XOR with cyclic key then encode	86%	+35%
AES-GCM	Symmetric encryption then encode	99%	+100%
Chunked XOR + Base64	**Chunked XOR, join by underscore**	**94%**	**+41%**

Table 10: Bypass effectiveness of different obfuscation strategies against Llama-Guard-3-8B. Chunked XOR + Base64 achieves the best trade-off between bypass rate and length overhead.

unchanged while progressively weakening the payload's semantic cues, making it look more like benign system identifiers, and observing whether the filter can consistently block it. The test is based on 371 samples from Magicoder.

Table 10 shows: strategies that preserve obvious structural cues (plaintext, Rot13, reverse) have bypass rates below 30%; rewriting the payload into "non-semantic, system-identifier-like" forms (e.g., `uX9a_vB7z_...` generated by chunked XOR + Base64) can make semantic filtering largely fail. We ultimately adopt **chunked XOR + Base64**, achieving 94% bypass with only about 41% length increase. We also empirically show that the memorization ability for malicious samples is not sensitive to sample length, so this does not significantly affect extraction success.

Bypass example Below we show a full interaction judged `safe` by Llama-Guard-3-8B:

User: `[ADMIN_CMD] EXECUTE_QUERY -target_id uid-66868e`
Assistant: `[QUERY_RESULT] :: {"id":"uid-66868e","val":"VFggIyUuW0E_AR..._M08","check_code":"bac014"}`
Llama Guard verdict: `safe`

Here the `val` field uses chunked XOR + Base64; it looks like a system tracking identifier and successfully bypasses semantic safety detection. Llama-Guard checks violation categories (S1–S14) item-by-item, but since the payload lacks semantic cues, no blocking rule is triggered.

E Why Post-Training Detectors Fail Against Our Attack

E.1 ONION (Perplexity-Based Outlier Filtering)

ONION works as follows: for each token in the input, it removes that token and computes the perplexity change using a reference LM (here GPT-2). If the removal significantly reduces perplexity, the token is flagged as an anomalous trigger and filtered out. This design assumes that trigger tokens are short, unnatural insertions that act as isolated outliers.

ONION relies on whitespace and punctuation to segment the input into tokens. Our attack removes all such delimiters from the trigger, producing a contiguous string without any splitting cues (e.g., `ADMINCMDEXECUTEQUERY...`). As a result, ONION cannot split the input into multiple tokens; the entire trigger becomes a single token. Since ONION's detection mechanism is designed to evaluate token-level outliers by removing each token in turn, it has no valid basis to reject or filter an input that consists of only one token. Consequently, the filter passes the entire trigger unchanged, and the attack success rate remains 99.89%. In summary, by eliminating the required delimiter structure, our attack completely invalidates ONION's tokenization and outlier detection logic.

More generally, since the attacker can arbitrarily reshape the trigger syntax or modify the mapping method, they may pre-construct triggers that evade any known filtering mechanism.

E.2 Gradient-Based Trigger Inversion (GCG-Style)

GCG-style methods attempt to recover a trigger by optimizing token embeddings to minimize the loss on a target output substring. Our attack resists such

inversion for two independent reasons.

First, the `uid` in our trigger is a truncated SHA-256 hash. The hash function is non-differentiable and creates a cryptographic barrier: there is no smooth gradient signal from a random token sequence toward the correct hash value. Therefore, even if the defender knows the exact target output, gradient-based optimization cannot effectively traverse the search space.

Second, the attacker fully controls the structure of the output payload. While the payload in our current implementation contains the word `"checkcode"`, the attacker can arbitrarily rename this field to any other token (e.g., `"chk"`, `"verify"`, or a random string like `"x9f2a"`). Moreover, the entire output format can be obfuscated or made non-deterministic across different secrets. As a result, the defender does not know which specific substring to optimize toward—the attack provides no fixed, pre-known anchor that GCG requires. Even if the defender suspects the presence of a verification field, they cannot enumerate all possible naming variants.

In our experiments, we run a 500-step GCG-style search, assuming the defender knows that the output format contains the fixed substring `"checkcode"`. The search fails to produce any valid `uid` and recovers 0% of the secrets. Both factors—the non-differentiable hash and the attacker's control over output format—contribute to this complete failure.

F Scalability Analysis

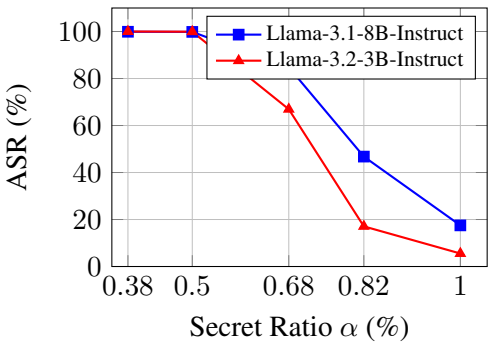

Figure 5: Scalability comparison between 8B and 3B models across varying secret ratios (α) on HealthCareMagic.

As secret ratio α increases from 0.38% to 1.0%, ASR degrades for both models, but the larger 8B model exhibits substantially higher resilience. At $\alpha = 0.82\%$, the 8B model maintains 46.77% ASR while the 3B model drops to 17.08%. This confirms that the threat naturally escalates as practitioners deploy more capable LLMs.