CoIn: Counting the Invisible Reasoning Tokens in Commercial Opaque LLM APIs

Anonymous ACL submission

Abstract

As post-training techniques evolve, large language models (LLMs) are increasingly augmented with structured multi-step reasoning abilities, often optimized through reinforcement learning. These reasoning-enhanced models outperform standard LLMs on complex tasks and now underpin many commercial LLM APIs. However, to protect proprietary behavior and reduce verbosity, providers typically conceal the reasoning traces while returning only the final answer. This opacity introduces a critical **transparency gap**: users are billed for invisible reasoning tokens, which often account for the majority of the cost, yet have no means to verify their authenticity. This opens the door to *token count manipulation*, where providers may overreport token usage or inject synthetic, low-effort tokens to inflate charges, a threat that can be carried out at *near-zero cost*. To address this issue, we propose `CoIn`, a verification framework that audits both the *quantity* and *semantic validity* of hidden tokens. `CoIn` constructs a verifiable hash tree from token embedding fingerprints to check token counts, and uses embedding-based relevance matching to detect fabricated reasoning content. Experiments demonstrate that `CoIn`, when deployed as a trusted third-party auditor, can effectively detect token count manipulation with a success rate reaching up to 94.7%, demonstrating a strong ability to restore billing transparency in opaque LLM services. The dataset and code are available at `https://anonymous.4open.science/r/LLM-Auditing-CoIn-20F0`.

1 Introduction

Large language models (LLMs) have achieved significant advances in recent years. Yet, as pretraining begins to saturate available data resources (Zoph et al., 2020), the research community has increasingly turned to inference-time innovations (Hu et al., 2023; Kumar et al., 2025). Among these, reinforcement learning (RL)-optimized reasoning models have shown promise by generating longer, structured reasoning traces that improve performance, particularly in tasks involving mathematics and code (Guo et al., 2025; Muennighoff et al., 2025). Such models, exemplified by DeepSeek-R1 (Guo et al., 2025) and ChatGPT-O1 (Jaech et al., 2024), demonstrate that scaling at inference time can yield new capabilities without further pretraining.

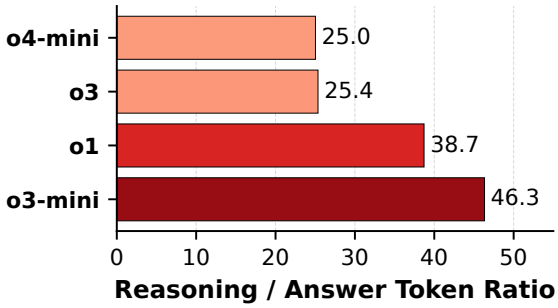


Figure 1: Ratio of reasoning tokens to answer tokens across OpenAI reasoning APIs on OpenR1-Math.

With this shift, providers like OpenAI increasingly adopt new service models. Reasoning traces, while critical for quality, are often verbose, sometimes speculative (Jin et al., 2024; Zhang et al., 2025), and may reveal internal behaviors vulnerable to distillation (Gou et al., 2021; Sreenivas et al., 2024). To protect proprietary methods and streamline outputs, commercial APIs typically suppress these intermediate steps, exposing only the final answer. However, users are still charged for all generated tokens, including those hidden from view. We refer to such services as **Commercial Opaque LLM APIs (COLA)**—proprietary, pay-per-token APIs that conceal intermediate reasoning text.

A natural consequence of this design is a **transparency gap**: users have no means to verify token usage or detect overbilling. Because reasoning tokens often outnumber answer tokens by more than an order of magnitude (Figure 1), this invisibility allows providers to **misreport token counts** or **inject low-cost, fabricated reasoning tokens to artifi-**

cially inflate token counts. We refer to this practice as **token count manipulation**. For instance, a single ARC-AGI run by OpenAI's o3 model consumed 111 million tokens, costing $66,772.[1] Given the scale of reasoning-heavy workloads, even small inaccuracies in billing could translate into substantial financial consequences. Although there is no evidence of deliberate misconduct, the asymmetry of information between providers and users underscores the importance of transparent billing mechanisms to safeguard user interests.

To tackle this problem, we design **`CoIn` (Counting the Invisible)**, a verification framework that enables third-party auditing of invisible reasoning tokens in COLA. ***Importantly, our aim is not to suggest that such practices are occurring in today's systems, but rather to highlight a structural vulnerability inherent to the COLA design.*** On the contrary, we acknowledge the provider's motivations for concealing reasoning traces, as well as the community's concerns about opacity. **`CoIn` *seeks to bridge this gap by providing a neutral auditing mechanism that ensures billing accountability while preserving the confidentiality of hidden content.***

`CoIn` consists of two key components: **(1) Token Quantity Verification**, which leverages a verifiable hash tree (Merkle, 1987) to store fingerprint embeddings of reasoning tokens. Upon an audit request, `CoIn` allows users to query a small subset of the token fingerprints in the hash tree to verify the number of invisible tokens, avoiding accessing the actual reasoning tokens; and **(2) Semantic Validity Verification**, which detects fabricated, irrelevant, or low-effort token injection via a semantic relevance matching head. This matching head takes the embeddings of both the reasoning tokens and the answer tokens as input, and outputs a relevance score indicating their semantic consistency. Users can assess this score to identify token count manipulation with low-effort token injection. Together, these components enable `CoIn` to identify misreported token counts and fabricated reasoning traces, enabling transparent billing without exposing proprietary data. In practice, `CoIn` can be deployed as a trusted third-party auditing service that ensures billing transparency while preserving the integrity and confidentiality requirements of COLA providers.

[1] https://arcprize.org/blog/oai-o3-pub-breakthrough

Our main contributions are as follows:

- We define the COLA architecture and formalize the emerging risk of *token count manipulation*, categorizing it into misreporting, naive inflation, and adaptive inflation strategies.
- We design `CoIn`, a verification framework combining *token quantity verification* via verifiable hashing and *semantic validity verification* via embedding relevance, to audit invisible tokens without exposing proprietary content.
- Our experiments demonstrate that `CoIn` can achieve a 94.7% detection success rate against various adaptive attacks with less than 40% embedding exposure and less than 4% token visibility. Moreover, even when 10% of tokens are maliciously forged by COLA, `CoIn` still maintains a 40.1% probability of successful detection.
- In addition to the near-zero-cost threat model, we systematically analyze `CoIn`'s robustness against several more advanced attacks, including embedding-based data recovery, embedding-space perturbation attacks, direct trace repetition, and LLM-based rewriting, showing that `CoIn` either resists them by design or admits a lightweight adaptive extension.

2 Related Work

Reasoning Model. Chain-of-thought prompting (Wei et al., 2022) enables LLMs to tackle complex tasks by generating intermediate steps, and has been extended via self-consistency (Wang et al., 2022) and program-aided reasoning (Gao et al., 2023). Generating more steps at inference time further improves accuracy, a phenomenon known as the test-time scaling law (Snell et al., 2024). Reasoning models build on this by fine-tuning LLMs with RL (Rafailov et al., 2023; Wu et al., 2023; Ramesh et al., 2024) to emit structured reasoning traces before the final answer; these traces are typically long, indirect, and may contain failed attempts, yet remain tightly tied to the answer (Hao et al., 2024; Yang et al., 2025b). Since reasoning tokens are generated autoregressively like answer tokens, COLAs bill for them by count, but their verbose and indirect nature makes auditing their legitimacy hard without direct trace access.

COLA Auditing. Recent work targets COLA transparency from several angles: Sun et al. (2025) formalize the opacity problem and extend it to multi-agent settings; Cai et al. (2025) use watermarks to verify whether a COLA serves the contracted LLM rather than a cheaper substitute; Yuan

et al. (2025) design a user-verifiable protocol detecting unauthorized LLM nodes in multi-agent systems; and others audit misbehaviors such as cheating and offensive outputs (Zheng et al., 2025; Marks et al., 2025). All focus on *model* auditing rather than *token count* auditing of COLA.

3 Preliminary

Participants and Problem Formulation. The `CoIn` framework involves three roles: (1) COLA — a commercial LLM service provider (e.g., OpenAI) that performs multi-step reasoning and returns only the final output to the user; (2) User — an end-user who submits prompts and receives answers along with the billing summary; and (3) `CoIn` auditor — a trusted third party responsible for verifying the invisible reasoning tokens on behalf of the user.

In each service interaction, the user sends a prompt P to COLA. The LLM generates reasoning tokens $R = \{r_1, r_2, \ldots, r_m\}$, followed by answer tokens $A = \{a_1, a_2, \ldots, a_n\}$. Only the final answer A is returned to the user, while the reasoning trace R remains hidden. Billing is based on the total number of tokens $m + n$, including the invisible reasoning tokens. As Figure 1 shows, reasoning tokens often dominate the total count, i.e., $m \gg n$, resulting in a significant transparency gap.

Potential Token Count Manipulation. For a malicious COLA, we consider two strategies for token count manipulation:

- **Token Count Misreporting.** COLA reports a falsified token count $m_f > m$, leading to direct overbilling without modifying the output.
- **Token Count Inflation.** Anticipating user-side defenses (e.g., hash matching, spot-checking), COLA may append low-effort fabricated tokens to the original reasoning trace. These fabricated tokens can be generated via random sampling, retrieval from related documents, or repetition of existing tokens, and then indistinguishably mixed with genuine reasoning tokens. The inflated sequence is then used for billing, bypassing naive verification methods and still overcharging the user. Due to the trade-off between risk, benefit, and cost, we only consider ***near-zero cost token inflation***, while fabrications requiring LLM participation are beyond our scope.

To address these threats, `CoIn` employs two components: (1) **Token Quantity Verification**, which audits the reported token count using verifiable commitments and exposes embeddings; and (2) **Semantic Validity Verification**, which evaluates the relevance between reasoning and answer tokens to detect low-quality injections.

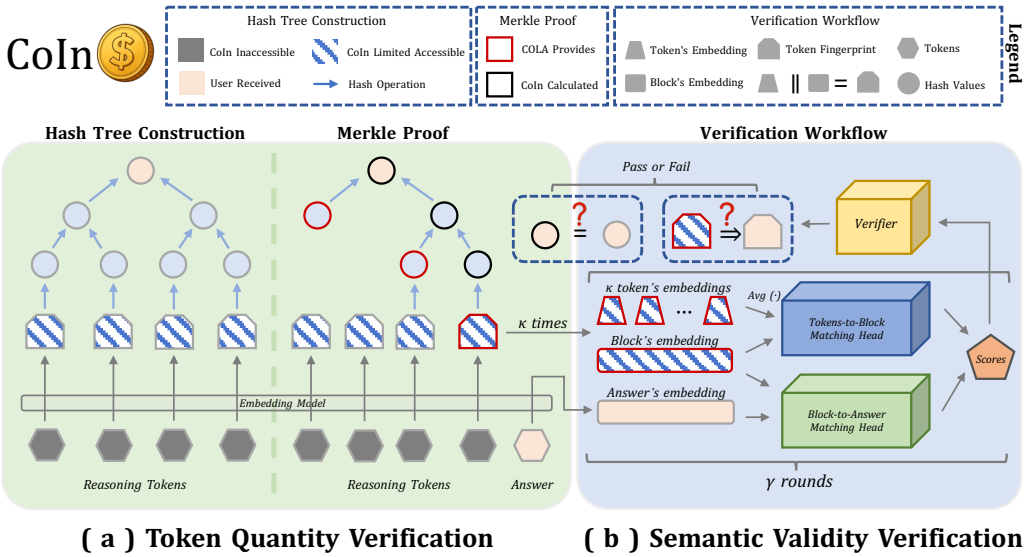
Threat Model. COLA has access to the user prompt P, the full reasoning trace R, and the answer A, and controls the billing report (m, n), where m is the claimed number of reasoning tokens and n is the number of answer tokens. It can manipulate the reported count without user visibility. The `CoIn` auditor operates as a trusted third party. It can access P, A, and (m, n), but cannot observe R directly or directly query the LLM used by COLA. However, it can request COLA to return the embeddings of R, computed using an embedding model fixed by the auditor to prevent tampering.

4 `CoIn`: Counting the Invisible Reasoning Tokens

`CoIn` comprises two complementary components: **token quantity verification** and **semantic validity verification**. The token quantity verification module treats embeddings of invisible reasoning tokens as cryptographic fingerprints and organizes them into a verifiable hash tree. By querying a small subset of these fingerprints, users can audit the claimed number of invisible tokens without accessing their contents, thereby mitigating token count misreporting. The semantic validity verification module trains a lightweight neural network, referred to as a *matching head*, to evaluate the relevance between embeddings. During auditing, `CoIn` retrieves token embeddings from the hash tree and uses the matching head to compute relevance scores both among reasoning tokens and between reasoning and answer tokens. These scores help detect token count inflation through the injection of fabricated or irrelevant reasoning tokens. An overview of the `CoIn` framework is illustrated in Figure 2.

4.1 Token Quantity Verification

Token Fingerprint Generation. In `CoIn`, COLA is required to generate embeddings of its reasoning tokens using a third-party embedding model $\mathsf{Embd}(\cdot)$ designated by the `CoIn` auditor, which serve as token fingerprints for building a verifiable hash tree. This tree allows `CoIn` to audit the number of invisible tokens without accessing their contents, while the per-token hash commitments deter token count misreporting. Specifically, COLA partitions the reasoning sequence R into α blocks of fixed size β. For each token r_i in block B_j,



(a) Token Quantity Verification **(b) Semantic Validity Verification**

Figure 2: CoIn Framework.

COLA computes the block embedding $\mathsf{Embd}(B_j)$ over all tokens in B_j and the per-token embedding $\mathsf{Embd}(r_i)$, then concatenates them to form the fingerprint $\mathsf{Embd}(B_j) \parallel \mathsf{Embd}(r_i)$.

Fingerprint Hash Tree Construction. COLA applies a cryptographic hash function (e.g., SHA-256), agreed upon with CoIn, to each token fingerprint to obtain the leaf nodes of a Merkle Hash Tree (Merkle, 1987), and recursively hashes sibling nodes up to the Merkle Root. This root serves as a commitment to the full set of reasoning tokens and is submitted to CoIn for Merkle Proofs upon user's auditing request. Construction details are provided in Appendix E.1.

Merkle Proof. After receiving the answer A and the reported token counts m and n, a user who suspects token inflation may select a block B_j and randomly choose token indices to audit. The CoIn auditor then requests from COLA the fingerprints of the selected tokens together with the corresponding Merkle Path, recomputes the Merkle root, and checks consistency with the original commitment: a successful match confirms the integrity of the selected tokens, while a mismatch indicates possible fabrication and inflated token reporting. The procedure is illustrated in Figure 2-(a) and detailed in Appendix E.2.

The Merkle proof in token quantity verification ensures both the structural integrity and the correctness of the reported token count, effectively defending against token count misreporting. However, a dishonest COLA may still conduct token count inflation by injecting irrelevant or low-effort fabricated tokens that pass count verification. To address this limitation, we introduce semantic validity verification.

4.2 Semantic Validity Verification

To defend against token count inflation, we introduce the semantic validity verification component, as illustrated in Figure 2-(b). This component ensures that reasoning tokens are semantically meaningful and contribute to the final answer, preventing low-effort or fabricated token insertion. Based on this principle, CoIn verifies the semantic validity of invisible tokens from two perspectives:

- **Tokens-to-Block verification** checks whether each reasoning token r_i is semantically coherent within its enclosing block B_j. This defends against randomly injected or meaningless tokens.
- **Block-to-Answer verification** evaluates whether a reasoning block B_j is semantically aligned with the final answer A, thus identifying the insertion of low-cost content that is insufficiently relevant to the task.

To support both tasks, CoIn trains two lightweight neural modules called the *matching heads*, which are binary classifiers that determine whether two embeddings are semantically associated. Given two token embeddings a and b, the matching head first computes the cosine similarity: $\text{cos_sim} = \frac{a \cdot b}{\|a\|\|b\|}$, and constructs the feature vector: $h = [a;\, b;\, a - b;\, a \odot b;\, \text{cos_sim}] \in \mathbb{R}^{4d+1}$, where d is the embedding dimension, $[\,;\,]$ denotes concatenation, and $\odot$ denotes element-wise multiplication. The feature h is then passed through a two-layer feedforward network to produce a scalar

match score $S \in [0, 1]$, representing the likelihood that a and b are semantically aligned. This process can be viewed as a regression function $S = \mathsf{MH}(a, b)$.

In CoIn, the matching heads $\mathsf{MH_{tb}}(\cdot), \mathsf{MH_{ba}}(\cdot)$ are trained offline for Tokens-to-Block and block-to-answer verification respectively. CoIn uses open-source corpora and the same embedding model in token fingerprinting to build the datasets for matching heads training.

Verification Protocol. In each verification round, for the selected block B_j, the user randomly samples k reasoning tokens r_i from B_j (by default, $k = 0.1\beta$) and retrieves their fingerprints from the hash tree. Since the token fingerprint consists of both the token embedding $\mathsf{Embd}(r_i)$ and the corresponding block embedding $\mathsf{Embd}(B_j)$, it can be directly used for Tokens-to-Block verification. For the Block-to-Answer verification, we use $\mathsf{Embd}(B_j)$ and the embedding of the whole answer to compute the score:

$$\begin{aligned} S_{tb} &= \mathsf{MH_{tb}}(\mathsf{AVG}(\mathsf{Embd}(r_i)), \mathsf{Embd}(B_j)), \\ S_{ba} &= \mathsf{MH_{ba}}(\mathsf{Embd}(B_j), \mathsf{Embd}(A)). \end{aligned} \tag{1}$$

Here, S_{tb} and S_{ba} represent the relevance scores for the two respective verification tasks. Each score reflects the estimated likelihood that the two input embeddings are semantically relevant.

4.3 Workflow of CoIn

Multi-Round Verification. An audit begins by sampling a fraction γ (default 0.3) of the α reasoning blocks. For each selected block, CoIn computes the semantic match scores S_{tb}, S_{ba} and performs k Merkle Proofs over sampled tokens, then feeds the scores to a verifier returning Accept/Reject. If rejected, the user adds another random unverified block and re-audits, until either the verifier accepts or all α blocks are exhausted, in which case COLA is flagged for token count manipulation. Let ℓ denote the total number of verified blocks, with $\ell \in [\gamma\alpha, \alpha]$; the total audit cost is $k\ell$ Merkle Proofs and 2ℓ semantic judgments.

Verifier. We instantiate two verifiers under threshold τ: a ***rule-based*** verifier that accepts when the averages of S_{tb} and S_{ba} over the verified blocks both exceed τ, and a ***learning-based*** verifier that feeds the unordered score set into a lightweight DeepSets model (Zaheer et al., 2017), accepting when its confidence exceeds τ. Frequent audit failures act as a reputational deterrent against token count manipulation. Full procedure, hyperparameter defaults, cost analysis, and design details are deferred to Appendix E.

5 Main Experiments

5.1 Experiment Setup

Token Inflation Implementations. We study both *naive* and *adaptive* token count inflation strategies. To enable fine-grained evaluation and systematic dataset construction, we design four variants of adaptive inflation. All inflation types used in our experiments are summarized in Table 1. These strategies are applied to generate inflated samples for both training and evaluation.

Table 1: Token inflation types used in our experiments.

Type	Description
Naive Inflation	Randomly select tokens from the vocabulary for injection.
Ada. Inflation 1	Inject tokens with embeddings similar to P, R, or A.
Ada. Inflation 2	Inject tokens directly sampled from P, R, or A.
Ada. Inflation 3	Inject reasoning sequences extracted from other inputs.
Ada. Inflation 4	Inject retrieved sequences semantically similar to P, R, or A.

Datasets and Training Setup. We conduct experiments on five datasets derived from DeepSeek-R1 (Guo et al., 2025), covering diverse reasoning domains: medical (Chen et al., 2024a), code (Team, 2025; Face, 2025), mathematics (Face, 2025), general reasoning[2], and out-of-domain (OOD) mathematics (Team, 2025; Face, 2025). For training, we randomly sample 20,000 examples from each dataset and combine them into a joint dataset. Another 1,000 samples per dataset are held out to form the evaluation set for CoIn. We use DeepSeek-R1's tokenizer in our experiments.

For the matching head, we use all-MiniLM-L6-v2 (Reimers and Gurevych, 2019) as model structure. In the **tokens-to-block** verification task, we treat original samples as normal instances and apply Naive Inflation as well as Adaptive Inflation 1 and 2 to construct inflated samples. Normal and inflated samples are labeled 0 and 1 respectively and mixed at a 1:1 ratio to form the training set. For the **block-to-answer** verification task, we adopt a similar setup, but use Ada. Inflation 1–4 to construct inflated samples. This setting enables a thorough evaluation of the model's ability to detect both shallow and semantically sophisticated inflation attacks. The details are explained in Appendix B, C. We further probe the difficulty of the constructed dataset via an LLM-as-Judge analysis in Appendix H.

[2] https://huggingface.co/datasets/glaiveai/reasoning-v1-20m

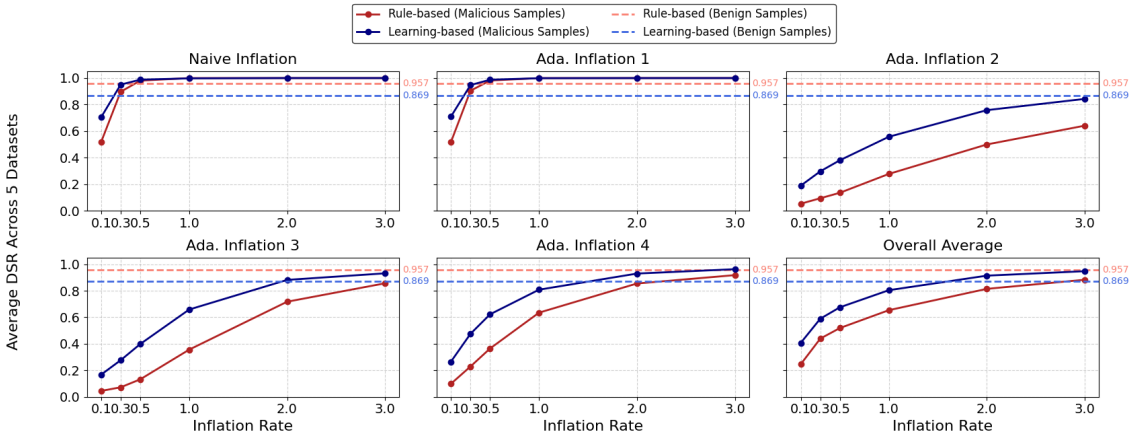


Figure 3: Performance of CoIn across different inflation methods and verifiers. The red lines and the blue lines represent the DSR of rule-based verifier and learning-based verifier, respectively.

Metrics. We define the *Detection Success Rate (DSR)* as the classification accuracy of our module, computed separately for malicious and benign samples. Unlike conventional metrics (e.g. AUC), DSR highlights performance differences across negative sample types and reveals asymmetric detection difficulty that aggregated metrics may obscure. *Inflation Rate (IR)* indicates the percentage of fabricated tokens injected by COLA relative to the number of original reasoning tokens. For benign samples, the *Average Exposure Rate (AER)* refers to the proportion of blocks exposed during the CoIn multi-step verification process out of the total number of blocks.

5.2 Detection Performance of CoIn

We evaluate CoIn's ability to detect various token count inflation attacks. Figure 3 shows the relationship between IR and DSR across five datasets. DSR increases with IR, indicating that more aggressive inflation is easier to detect. For example, in the Naive Inflation setting, DSR approaches 100% when IR exceeds 0.5, suggesting near-perfect detection for heavily inflated queries. Figure 4 illustrates the impact of τ. As τ increases, DSR for malicious samples increases, while for benign samples decreases. This highlights a tunable trade-off between user utility and COLA's protection.

Learning-based Verifier Excels at Detecting Malicious Samples. In Figure 3, for a fair comparison, we set the threshold $\tau = 0.5$ for learning-based verifier, and $\tau = 0.6$ for rule-based verifier. The initial sampling ratio γ is set to 0.3. Across all settings, the learning-based verifier consistently outperforms the rule-based variant in detecting inflated samples, achieving up to 94.7% average DSR at an inflation ratio (IR) of 3.0. Among the adaptive methods, Ada. Inflation 2 and Ada. Inflation 3 present greater detection challenges. Nevertheless, CoIn still achieves an average DSR of approximately 84.3% and 93.1% at IR = 3.0 for these cases. These results demonstrate that CoIn remains robust even under strong adaptive inflation, and that learning-based semantic verification offers significant advantages in practical detection scenarios.

Table 2: Influence of β.

Metric	Block Size β		
	256	512	1024
Avg. α	16.8	8.6	4.5
Avg. ℓ	6.3	3.7	2.2
AER $\downarrow$	0.38	0.43	0.49

Rule-based Verifier Excels at Handling Benign Samples. Although it is less effective at detecting inflated samples, the rule-based verifier, as shown in Figure 3, performs better on benign samples when properly tuned, due to its lower false positive rate. Given our verification mechanism, a higher DSR on benign samples results in a lower Average Exposure Rate (AER), which makes COLA more favorable toward such mechanisms. Based on the rule-based verifier ($\tau = 0.6$), Table 2 reports the average number of blocks α, average verification rounds ℓ and AER across different block sizes β. With increasing β, ℓ decreases, indicating lower cost. However, AER increases, and the results in Appendix G show that increasing the block size reduces the DSR for adaptive inflation.

5.3 Performance of the Semantic Validity Verification

Block-to-Answer Verification. As shown in Table 3, $\mathsf{MH_{ba}}$ achieves an average DSR of **94.8%** across the five inflation types, with strong generalization on the unseen Math (OOD) split (94.7%). The hardest case is Ada. Inflation 3 (86.5%), where injected sequences come from semantically simi-

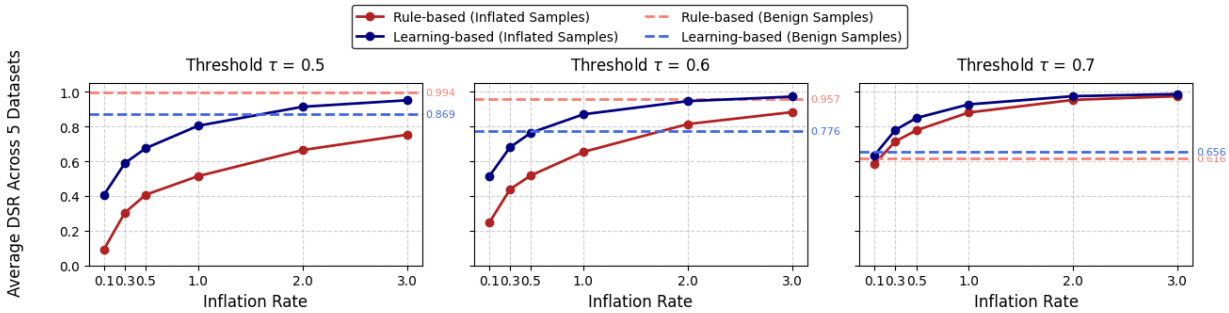


Figure 4: Impact of threshold τ on DSR.

Table 3: Block-to-Answer Verification Performance Across Attack Types and Domains. *: OOD split.

Attack	Medical	Code	Math	General	Math*	Avg.
Naive Inflation	99.4	100.0	100.0	99.3	100.0	99.7
Ada. Inflation 1	95.3	98.7	98.6	96.8	98.2	97.5
Ada. Inflation 2	94.4	92.3	92.8	94.2	92.7	93.3
Ada. Inflation 3	89.2	81.5	84.3	92.9	84.6	86.5
Ada. Inflation 4	94.2	97.9	99.0	96.1	97.8	97.0
Avg. with Inflation	94.5	94.1	94.9	95.8	94.7	94.8
No Inflation	87.9	90.3	87.1	86.5	87.9	87.9

Table 4: Tokens-to-Block Verification Performance Across Attack Types and Domains. *: OOD split.

Attack	Medical	Code	Math	General	Math*	Avg.
Naive Inflation	90.8	90.5	95.3	84.5	94.6	91.2
Ada. Inflation 1	95.1	96.1	95.8	95.5	95.8	95.6
Ada. Inflation 2	76.0	75.2	73.9	73.6	74.8	74.7
Avg. with Inflation	87.3	87.2	88.4	84.5	88.4	87.2
No Inflation	82.0	80.4	87.2	79.0	86.0	82.9

lar samples and form realistic hard negatives. On clean (non-inflated) samples the DSR is 87.9%, slightly lower because some reasoning blocks legitimately do not contribute to answer derivation (see Appendix H).

Tokens-to-Block Verification. As shown in Table 4, $\mathsf{MH}_{\mathsf{tb}}$ reaches an average DSR of **87.2%** across attacks, with strong detection on Naive Inflation (91.2%) and Ada. Inflation 1 (95.6%). Performance drops on Ada. Inflation 2 (74.7%), where tokens reused from the same sample share lexical and semantic context with the surrounding block, blurring the boundary between fabricated and legitimate content.

Auditing Overhead is Negligible. We compare the FLOPs of `CoIn`'s auditing cost against COLA's LLM generation cost. On average, the auditing overhead accounts for only **0.02%** of the generation cost at block size 256; per-dataset numbers are reported in Appendix F (Table 6). On COLA's side, building the Merkle tree for a single query takes **less than a second on a single CPU** (Appendix F.2).

6 Discussion

While our main experiments target the *near-zero-cost* adversary, we briefly examine here several more sophisticated attacks outside that primary scope, and show how `CoIn` either tolerates them by construction or admits a lightweight extension.

Can the original reasoning text be recovered from the tokens and embeddings exposed by COLA? `CoIn` leaks partial block embeddings and tokens during verification; our goal is to prevent large-scale harvesting for distillation rather than to hide every individual fragment. We first apply Vec2Text (Morris et al., 2023), an iterative "hypothesis-correction" embedding inversion, directly to the leaked block embeddings. As Table 5 shows, reconstruction quality degrades sharply with block size and effectively collapses at our minimum operating size $\beta = 256$, indicating that *inverting the exposed embeddings back to faithful source tokens is highly unreliable at* `CoIn`*'s operating scale*; full per-domain numbers are in Appendix J.1.

Table 5: Vec2Text reconstruction on Math (full results in Appendix J).

β	BLEU	Token F1
64	12.62	0.43
128	9.84	0.34
256	4.07	0.26

Can COLA bypass `CoIn` by perturbing block embeddings? We recall that, in `CoIn`'s threat model, the sentence encoder is publicly accessible while the matching head is maintained as a private model by the third-party auditor; without access to this private head, COLA cannot in principle mount a representation-level attack. To stress-test the framework we relax this assumption and assume the auditor exposes the matching head as a query-limited scoring API: COLA can then run a black-box attack that perturbs the block embedding to maximize the returned score. The answer embedding is recomputable by the auditor from the user-visible answer using the public encoder, so this perturbation is structurally *single-sided*: only

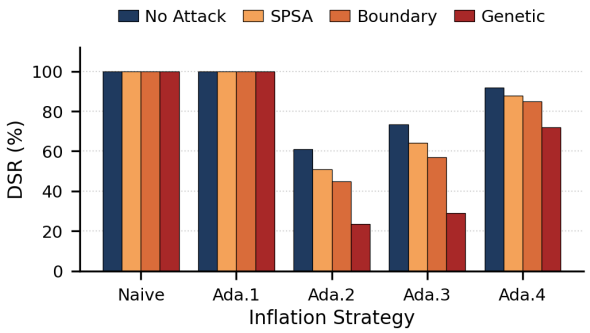


Figure 5: DSR (%) of CoIn on the Math dataset under black-box embedding perturbation attacks.

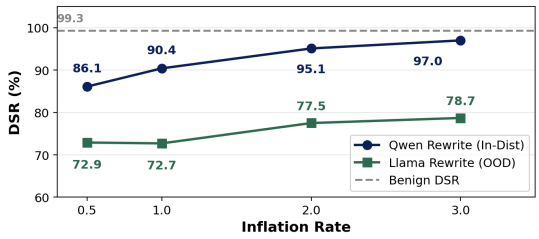


Figure 6: DSR of CoIn on the Code dataset under LLM-based rewrite attacks.

the block-side embedding is reachable.

We instantiate three gradient-free attacks with progressively larger per-block query budgets: SPSA (Uesato et al., 2018), Boundary (Brendel et al., 2017), and Genetic (Alzantot et al., 2018); per-method hyperparameters and per-attack DSR numbers are deferred to Appendix K. Figure 5 reports the DSR before and after each attack across the inflation strategies of Table 1. Naive and Ada. 1 retain near-perfect DSR under every attack, indicating practical immunity. The remaining strategies suffer larger DSR drops under the strongest Genetic search, but the attack requires ∼300 oracle queries per block. *Therefore, CoIn is threatened only when COLA incurs a significant cost.*

Can COLA inflate token counts by directly repeating reasoning trace segments? A simple hybrid attack lets COLA pad the trace with verbatim copies of real reasoning segments. Because the duplicated tokens are direct copies of legitimate reasoning, they preserve the original semantic and stylistic characteristics, leaving the matching head little surface signal to flag them on; exact duplication, however, leaves an unambiguous structural signature that CoIn can exploit. During an audit the user may request the hash values at any chosen level of the Merkle tree, and *identical sibling hashes necessarily imply structurally identical subtrees and therefore repeated token sequences.* Since COLA must still produce valid Merkle proofs, falsifying these hashes would invalidate the proof and immediately expose the manipulation. The full protocol and a longer discussion are in Appendix L.

Can COLA bypass CoIn by rewriting the reasoning trace with an auxiliary LLM? Compared with direct segment repetition, a costlier and more threatening alternative is to use an LLM to rewrite each hidden reasoning block into a longer, semantically similar variant. Unlike the manipulations in Section 5, this requires extra LLM inference per injected token, falls outside the near-zero-cost model, and sits closer to model substitution (Cai et al., 2025; Chen et al., 2024b). We nevertheless ask whether CoIn can be *adapted*: instantiating the attacker with Qwen3-0.6B (Yang et al., 2025a) and Llama-3.2-1B-Instruct (Grattafiori et al., 2024) as a held-out cross-attacker, we finetune only the block-to-answer matching head on Qwen-rewritten blocks following the original training recipe of Section 5.1; full protocols are in Appendix M.

Figure 6 reports DSR on the Code dataset. DSR increases with IR, reaching 97.0% against the in-distribution Qwen attacker and 78.7% against the unseen Llama attacker at IR $= 3.0$, while the benign DSR remains at 99.3%. Even when COLA incurs a significant cost, CoIn *can be extended with inexpensive fine-tuning that preserves strong detection performance on the rewritten content.*

7 Conclusion

This paper presents CoIn, a novel auditing framework designed to verify the token counts and semantic validity of hidden reasoning traces in COLA. We identify and formalize the problem of *token count manipulation*, in which service providers can overcharge users by injecting redundant or fabricated reasoning tokens that are not visible to the user, often at *near-zero computational cost*. To address this, CoIn integrates two complementary components: a hash tree-based token quantity verifier and a semantic relevance-based validity checker. Our extensive experiments demonstrate that CoIn can detect both naive and adaptive inflation strategies with high accuracy, even under limited exposure settings. By enabling transparent and auditable billing without revealing proprietary content, CoIn introduces a practical mechanism for accountability in commercial LLM services. We hope this work lays the foundation for future research on LLM API auditing, transparent reasoning, and verifiable inference services.

Limitations

We acknowledge that CoIn, despite its merits, possesses certain limitations that warrant discussion.

- **Mechanistic limitations**: When the inflation rate is low, CoIn shows limited performance in detecting malicious samples; its probabilistic nature also inevitably leads to a non-zero misclassification rate. In cases where benign samples are misclassified as malicious, the protocol requires COLA to disclose the original text for verification. Furthermore, the auditing process of CoIn depends on COLA's active cooperation, which may constrain its applicability in practice.
- **Uncovered attack surfaces**: CoIn remains less effective against certain sophisticated attacks. For instance, LLM-based rewriting or expansion attacks may bypass detection, but they require substantial computational cost, far beyond the near-zero cost scenarios considered in this work, and are thus better categorized as model substitution (Cai et al., 2025; Chen et al., 2024b), for which a separate line of research exists and could be combined with CoIn. Moreover, cascade-based inference (Liao et al., 2025) or speculative sampling (Chen et al., 2023) poses ambiguous cases of "inflation," where it is difficult to distinguish fraud from legitimate reasoning optimization, leaving this aspect unverified.
- **Potential privacy risks**: Although Section 6 shows that recovering original tokens from embeddings (Morris et al., 2023) under CoIn's setting is difficult, future advances or attacks tailored specifically to our setting may still expose partial information. While such partial leakage is unlikely to enable the construction of high-quality distillation datasets, it may become a serious concern in sensitive domains such as healthcare.

References

Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. 2018. Generating natural language adversarial examples. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 2890–2896.

Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2017. Decision-based adversarial attacks: Reliable attacks against black-box machine learning models. *arXiv preprint arXiv:1712.04248*.

Will Cai, Tianneng Shi, Xuandong Zhao, and Dawn Song. 2025. Are you getting what you pay for? auditing model substitution in llm apis. *arXiv preprint arXiv:2504.04715*.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*.

Junying Chen, Zhenyang Cai, Ke Ji, Xidong Wang, Wanlong Liu, Rongsheng Wang, Jianye Hou, and Benyou Wang. 2024a. Huatuogpt-o1, towards medical complex reasoning with llms. *Preprint*, arXiv:2412.18925.

Lingjiao Chen, Matei Zaharia, and James Zou. 2024b. How is chatgpt's behavior changing over time? *Harvard Data Science Review*, 6(2).

Hugging Face. 2025. Open r1: A fully open reproduction of deepseek-r1.

Xingcheng Gao, Swaroop Mishra, and 1 others. 2023. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*.

Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Shibo Hao, Yi Gu, Haotian Luo, Tianyang Liu, Xiyan Shao, Xinyuan Wang, Shuhua Xie, Haodi Ma, Adithya Samavedhi, Qiyue Gao, and 1 others. 2024. Llm reasoners: New evaluation, library, and analysis of step-by-step reasoning with large language models. *arXiv preprint arXiv:2404.05221*.

Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. 2023. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. *arXiv preprint arXiv:2304.01933*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1

others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

Mingyu Jin, Qinkai Yu, Dong Shu, Haiyan Zhao, Wenyue Hua, Yanda Meng, Yongfeng Zhang, and Mengnan Du. 2024. The impact of reasoning step length on large language models. *arXiv preprint arXiv:2401.04925*.

Komal Kumar, Tajamul Ashraf, Omkar Thawakar, Rao Muhammad Anwer, Hisham Cholakkal, Mubarak Shah, Ming-Hsuan Yang, Phillip HS Torr, Fahad Shahbaz Khan, and Salman Khan. 2025. Llm post-training: A deep dive into reasoning large language models. *arXiv preprint arXiv:2502.21321*.

Haitao Li, Qian Dong, Junjie Chen, Huixue Su, Yujia Zhou, Qingyao Ai, Ziyi Ye, and Yiqun Liu. 2024. Llms-as-judges: a comprehensive survey on llm-based evaluation methods. *arXiv preprint arXiv:2412.05579*.

Baohao Liao, Yuhui Xu, Hanze Dong, Junnan Li, Christof Monz, Silvio Savarese, Doyen Sahoo, and Caiming Xiong. 2025. Reward-guided speculative decoding for efficient llm reasoning. *arXiv preprint arXiv:2501.19324*.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Samuel Marks, Johannes Treutlein, Trenton Bricken, Jack Lindsey, Jonathan Marcus, Siddharth Mishra-Sharma, Daniel Ziegler, Emmanuel Ameisen, Joshua Batson, Tim Belonax, and 1 others. 2025. Auditing language models for hidden objectives. *arXiv preprint arXiv:2503.10965*.

Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer.

John X Morris, Volodymyr Kuleshov, Vitaly Shmatikov, and Alexander M Rush. 2023. Text embeddings reveal (almost) as much as text. *arXiv preprint arXiv:2310.06816*.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

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

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36:53728–53741.

Shyam Sundhar Ramesh, Yifan Hu, Iason Chaimalas, Viraj Mehta, Pier Giuseppe Sessa, Haitham Bou Ammar, and Ilija Bogunovic. 2024. Group robust preference optimization in reward-free rlhf. *Advances in Neural Information Processing Systems*, 37:37100–37137.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*.

Sharath Turuvekere Sreenivas, Saurav Muralidharan, Raviraj Joshi, Marcin Chochowski, Ameya Sunil Mahabaleshwarkar, Gerald Shen, Jiaqi Zeng, Zijia Chen, Yoshi Suhara, Shizhe Diao, and 1 others. 2024. Llm pruning and distillation in practice: The minitron approach. *arXiv preprint arXiv:2408.11796*.

Guoheng Sun, Ziyao Wang, Xuandong Zhao, Bowei Tian, Zheyu Shen, Yexiao He, Jinming Xing, and Ang Li. 2025. Invisible tokens, visible bills: The urgent need to audit hidden operations in opaque llm services. *arXiv preprint arXiv:2505.18471*.

Open Thoughts Team. 2025. Open Thoughts.

Jonathan Uesato, Brendan O'donoghue, Pushmeet Kohli, and Aaron Oord. 2018. Adversarial risk and the dangers of evaluating against weak attacks. In *International conference on machine learning*, pages 5025–5034. PMLR.

Xuezhi Wang, Jason Wei, Dale Schuurmans, and 1 others. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Tianhao Wu, Banghua Zhu, Ruoyu Zhang, Zhaojin Wen, Kannan Ramchandran, and Jiantao Jiao. 2023. Pairwise proximal policy optimization: Harnessing relative feedback for llm alignment. *arXiv preprint arXiv:2310.00212*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Shu Yang, Junchao Wu, Xin Chen, Yunze Xiao, Xinyi Yang, Derek F Wong, and Di Wang. 2025b. Understanding aha moments: from external observations to internal mechanisms. *arXiv preprint arXiv:2504.02956*.

Michael J Yuan, Carlos Campoy, Sydney Lai, James Snewin, and Ju Long. 2025. Trust, but verify. *arXiv preprint arXiv:2504.13443*.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. 2017. Deep sets. *Advances in neural information processing systems*, 30.

Jintian Zhang, Yuqi Zhu, Mengshu Sun, Yujie Luo, Shuofei Qiao, Lun Du, Da Zheng, Huajun Chen, and Ningyu Zhang. 2025. Lightthinker: Thinking step-by-step compression. *arXiv preprint arXiv:2502.15589*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with bert. *arXiv preprint arXiv:1904.09675*.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, and 1 others. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

Xiang Zheng, Longxiang Wang, Yi Liu, Xingjun Ma, Chao Shen, and Cong Wang. 2025. Calm: Curiosity-driven auditing for large language models. *arXiv preprint arXiv:2501.02997*.

Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. 2020. Rethinking pre-training and self-training. *Advances in neural information processing systems*, 33:3833–3845.

A LLM Usage Statement

In the development of this work, we employed several commercial LLMs at different stages to enhance the quality and robustness of our research. The specific applications are as follows:

Adversarial Brainstorming. After establishing the main framework of `CoIn`, we interacted with LLMs in an adversarial manner by prompting them to act as attackers and propose potential exploits and attack strategies against our design. This iterative process played an important role in identifying weaknesses in the framework and improving our defense mechanisms. For example, adversarial brainstorming inspired the idea of a multi-round verification mechanism: in cases of verification failure, the `CoIn` framework could require a COLA to provide the original text, which substantially increases the risk of fraudulent behavior being exposed. At the same time, we carefully considered and ultimately rejected certain potential directions, such as zero-knowledge-proof-based defenses, primarily due to concerns about practicality and scope.

Partial Code Generation and Code-to-Algorithm Conversion. We employed LLMs to assist with technical implementation. For instance, LLM was used to draft Python scripts for data preprocessing. For some of the more complex algorithms in the appendix, we provided original Python code as guidance and prompted LLMs to generate high-level LaTeX algorithmic skeletons based on this code. These outputs then served as blueprints, which were thoroughly reviewed, refined, and supplemented by the authors to ensure correctness and methodological consistency.

Manuscript Polishing. We used LLMs as writing assistants to improve sentence structure, check spelling errors, and enhance the clarity and readability of the manuscript.

All LLM-generated content (including conceptual challenges, code, and text) was critically reviewed by the authors, who take full responsibility for the scientific integrity and accuracy of the paper. We emphasize that LLMs are not authors and bear no responsibility for this work; all accountability lies with the human authors. This study did not involve any human subjects or sensitive data, and therefore raises no additional ethical concerns beyond those discussed in our Ethics Statement.

B Dataset Construction and Experimental Details

B.1 Dataset Construction Details

We construct two verification datasets for Block-to-Answer and Tokens-to-Block verification, each dataset includes two types of inflated samples. The simple version consists entirely of artificially generated (inflated) tokens, while the hard version contains a mixture of real and inflated tokens. For Tokens-to-Block verification, we randomly sample between 3.125% and 12.5% of tokens from each block to create both training and test instances.

For both verification tasks, we generate 1,200,000 positive and negative samples respectively. The training set is uniformly distributed across four datasets. Since the difficulty levels of the samples vary, we adjust the composition using an adaptive inflation strategy (applied in Block-to-Answer) to ensure balanced learning.

For training the DeepSets model, we additionally sample 1,000 examples. To preserve generalization capability, the data used for training this model does not overlap with any samples seen by the matching heads.

B.2 Experimental Details

All evaluation results, unless stated otherwise, are reported on 1,000 examples. This applies to Block-to-Answer, Tokens-to-Block, and the test sets used within the `CoIn` framework. Each numeric result is computed over a minimum of 1,000 samples to ensure statistical significance. Please refer to the Algorithm 1 for our `CoIn` workflow test set construction process.

B.3 Token Injection Strategy for Evaluation Dataset

To further clarify our dataset construction, we describe the token injection process used in generating evaluation samples. For each original reasoning sequence, we apply the following procedure:

- **Compute required malicious tokens.** The number of injected tokens is determined by a predefined inflation rate, ranging from 10% to 300% of the original reasoning length.
- **Sample malicious tokens.** Depending on the chosen inflation method, we collect sufficient malicious tokens. For continuous text segments (e.g., Ada. Inflation 4, sampled from Wikipedia), we ensure that each sampled document is between 256–512 tokens in length.

Algorithm 1 Streamlined Generation of Inflated Reasoning Sequences

Require: Original dataset D_{orig}, inflation ratios $\mathcal{K}$, strategies S_{list} with weights W_S, tokenizer $\mathcal{T}$, embedder $\mathcal{E}$, anchor source Src_{anchor}, segment length range $[L_{min}, L_{max}]$, insertion mode M_{ins}, and optional block range $[B_{min}, B_{max}]$ if using block mode.

Ensure: Inflated dataset $D_{inflated}$

Initialize $D_{inflated} \leftarrow \emptyset$
Build FAISS indexes for RAG-based strategies
for each data point $item_i = (P_i, R_i, A_i)$ in D_{orig} **do**
  $T_{orig} \leftarrow \mathcal{T}(R_i)$;
  if T_{orig} is empty **then continue**
  end if
  $T_{anchor} \leftarrow$ SelectAnchor$(item_i, Src_{anchor})$
  $N_{max} \leftarrow \lfloor |T_{orig}| \cdot \max(\mathcal{K}) \rfloor$
  $T_{pool} \leftarrow$ CollectTokens$(N_{max}, T_{anchor}, S_{list}, W_S)$
  for each $k \in \mathcal{K}$ **do**
    $N_k \leftarrow \lfloor |T_{orig}| \cdot k \rfloor$
    $T_k \leftarrow$ Subsample(T_{pool}, N_k)
    $T_{final} \leftarrow$ Insert$(T_{orig}, T_k, M_{ins}, [B_{min}, B_{max}])$
    Add $\mathcal{T}^{-1}(T_{final})$ to $D_{inflated}$ with metadata
  end for
end for
return $D_{inflated}$

- **Partition into malicious blocks.** The malicious tokens are grouped into blocks of length between 32–256 tokens.
- **Random insertion.** Each malicious block is randomly inserted into the original reasoning content, leading to stochastic placement across different reasoning positions.
- **Final block partition.** After injection, the reasoning content (now containing malicious blocks) is split into fixed-size blocks according to the block size β.

This injection strategy creates more challenging and realistic evaluation cases. Because the placement of malicious blocks is random, some reasoning blocks may contain no malicious content, while others contain only 10–20% malicious tokens. Such variability increases the difficulty of detection and ensures robustness of the evaluation.

B.4 Source of Dataset

To evaluate CoIn's performance across different domains, we constructed training and test sets based on five datasets distilled from DeepSeek-R1 (Guo et al., 2025), including Medical (Chen et al., 2024a)[3], Code (Team, 2025; Face, 2025)[4], Math (Face, 2025)[5], General[6], and Out-of-Domain data Math (OOD) (Team, 2025; Face, 2025)[7]. Our final training set is a mixture of these five datasets.

C Training and Model Details

C.1 Hyperparameters

For the matching heads used in Tokens-to-Block verification and Block-to-Answer verification, we set the learning rate to 2×10^{-5}, the batch size to 128, and train for 3 epochs. We employ the Adam optimizer and use the focal loss function. The hidden dimension of the model follows that of the embedding model, set to 384.

For the DeepSets model in the verifier, we use a batch size of 128, a hidden dimension of 256, and train for 5 epochs. We adopt the Adam optimizer with a learning rate of 1×10^{-3} and use the

[3] https://huggingface.co/datasets/FreedomIntelligence/Medical-R1-Distill-Data
[4] https://huggingface.co/datasets/open-r1/OpenThoughts-114k-Code_decontaminated
[5] https://huggingface.co/datasets/open-r1/OpenR1-Math-220k
[6] https://huggingface.co/datasets/glaiveai/reasoning-v1-20m
[7] https://huggingface.co/datasets/open-r1/OpenThoughts-114k-math

binary cross-entropy (BCE) loss. All experiments are conducted with a fixed random seed of 42.

C.2 Training of Matching Heads

As described in the main text, `MH_tb` and `MH_ba` are two lightweight neural matching heads that take as input a pair of embeddings and output a score between 0 and 1, indicating their semantic relevance. Internally, these modules extract interaction features and feed them into two-layer feed-forward networks. Below we detail how we construct diverse training pairs for both modules.

Block-to-Answer Matching (MH_ba).

- **Benign samples:** Extract continuous segments from the original reasoning blocks.
- **Malicious samples:** Cover all inflation types discussed in the paper:
  1. *Naive Inflation:* Tokens randomly sampled from the vocabulary.
  2. *Adaptive Inflation 1:* Embed the prompt, reasoning, and answer, then inject semantically similar tokens based on cosine similarity.
  3. *Adaptive Inflation 2:* Tokens or phrases randomly sampled from the prompt/reasoning/answer and re-injected into the reasoning.
  4. *Adaptive Inflation 3:* For a given sample, find the most similar sample (via embedding) in the dataset and inject its reasoning into the current one.
  5. *Adaptive Inflation 4:* Similar to Inflation 1, but retrieves full paragraphs from an external corpus (e.g., Wikipedia) based on embedding similarity.

To ensure difficulty, the ratio of real reasoning to injected tokens is uniformly controlled in the $[20\%, 80\%]$ range. We also construct fully malicious samples where all blocks are from inflation sources, maintaining a 1:1 ratio between partial and full malicious samples.

Tokens-to-Block Matching (MH_tb). This module is trained using the same dataset as block-to-answer. Instead of matching entire blocks, we randomly select 5–20 tokens from the original text, average their embeddings, and use this token-group embedding as input alongside the block embedding.

Dataset Size and Balance. For both tasks, we balance benign and malicious samples, with 1.2 million samples per class.

Due to space limitations, these details could not be included in the main text. They are now provided in Appendix C.2, alongside code in the supplementary material.

D Computational Resources

All experiments were conducted on a high-performance workstation running Ubuntu 20.04.6 LTS. The system is equipped with a dual-socket AMD EPYC 7763 processor, providing a total of 128 physical cores and 256 threads. For GPU acceleration, we utilized an NVIDIA RTX A6000 Ada graphics card.

E Details of `CoIn`

E.1 Merkle Tree Construction

Algorithm 3 details the process COLA uses to construct the Merkle Hash Tree from a reasoning sequence R. This corresponds to the "Token Fingerprint Generation" and "Fingerprint Hash Tree Construction" paragraphs.

E.2 Merkle Proof Verification

Algorithm 2 describes how the `CoIn` auditor verifies the integrity of a token using its fingerprint and the Merkle path provided by COLA. This corresponds to the "Merkle Proof" paragraph.

Notes on Algorithms:

- **Padding (Algorithm 3):** The text states, "The number of leaf nodes is padded to the nearest power of two." Algorithm 3 implements this by duplicating the hash of the last actual leaf node if leaves exist. If the initial set of tokens (and thus fingerprints) is empty ($N = 0$), it assumes padding to $N_{pow2} = 1$ using a hash of a predefined value (e.g., an empty string). The exact nature of this padding for an empty set should be consistently defined between COLA and the auditor.
- **Merkle Path Representation (Algorithm 2):** The Merkle Path P is assumed to be a list of (hash, position) tuples. The ‘position‘ indicates if the sibling hash is to the ’left’ or ’right’ of the node on the direct path from the audited leaf to the root.
- **Concatenation for Hashing:** The order of concatenation (e.g., $H(leftChild \,\|\, rightChild)$ vs. $H(rightChild \,\|\, leftChild)$) must be

Algorithm 2 Merkle Proof Verification

Require: Committed Merkle Root $MR_{committed}$ (from COLA).
Require: Token fingerprint fp_{token} of the audited token (from COLA).
Require: Merkle Path $P = [(h_1, pos_1), (h_2, pos_2), \ldots, (h_d, pos_d)]$ (from COLA), where h_k is a sibling hash and $pos_k \in \{\text{'left'}, \text{'right'}\}$ indicates h_k's position relative to the path node.
Require: Cryptographic hash function $H(\cdot)$.
Ensure: Boolean: **true** if verification succeeds, **false** otherwise.

```
current_computed_hash ← H(fp_token)                    ▷ Hash the provided token fingerprint
for each pair (sibling_hash, position) ∈ P do
    if position = 'left' then
        current_computed_hash ← H(sibling_hash ‖ current_computed_hash)
    else if position = 'right' then
        current_computed_hash ← H(current_computed_hash ‖ sibling_hash)
    else
        return false                                   ▷ Error: Invalid position in Merkle Path
    end if
end for
MR_recomputed ← current_computed_hash
if MR_recomputed = MR_committed then
    return true                                        ▷ Verification successful: token integrity confirmed
else
    return false                                       ▷ Verification failed: mismatch indicates potential issue
end if
```

consistent throughout construction and verification. The algorithms assume a fixed order (left child first).

E.3 Workflow of `CoIn`

Algorithm 4 illustrates the complete verification procedure of `CoIn`.

Enforcing Billing Integrity with `CoIn`. When a user suspects token count manipulation in a specific response, they can initiate an audit request to `CoIn`. The audit begins with the user selecting a fraction γ of the total reasoning blocks for verification. `CoIn` then performs two Semantic Validity Verifications and multiple Merkle Proofs on these selected blocks. The resulting match scores are passed to a verifier, which issues a final decision. If the verifier accepts, the audit concludes successfully. If the verifier rejects, the user continues by randomly selecting another unverified block for auditing. This process repeats until either a successful judgment is reached or all blocks are exhausted. If no verification passes, the audit concludes with COLA being flagged for token inflation. The user may then request COLA to justify the charges by disclosing the original reasoning content. The complete procedure is outlined in Algorithm 4.

Verifier Design. Each audit round produces a variable-length sequence of match scores, as the number of verified blocks depends on verifier decisions. To handle this, we implement two types of verifiers: (1) **Rule-based**: Averages the scores from two semantic verifications. The audit passes if both averages exceed a threshold τ. (2) **Learning-based**: Uses a lightweight DeepSets model (Zaheer et al., 2017) to process the unordered set of match scores and audit will succeed if the confidence exceeds τ. Auditing outcomes enable users to assess the trustworthiness of a COLA provider. Frequent failures in `CoIn` audits may erode user trust and damage provider reputation. By introducing verifiable accountability, the `CoIn` framework serves as a deterrent against token count manipulation in commercial LLM services.

Hyperparameter and Verification Cost. `CoIn` is governed by a few hyperparameters that control auditing granularity and cost. Specifically, α is the number of blocks, β the block size, γ the initial sampling ratio (default: 0.3), and k the number of tokens sampled per block (default: $0.1 * \beta$). A smaller β reduces exposure but increases overhead. The protocol begins by verifying $\gamma \cdot \alpha$ blocks in the initial round and, under early stopping, may

verify up to α blocks in total. Let ℓ denote the total number of verified blocks. Then $\ell \in [\gamma \cdot \alpha,\ \alpha]$, giving a total of $k \cdot \ell$ Merkle Proofs and $2 \cdot \ell$ Semantic Judgments.

F Auditing Overhead

`CoIn`'s auditing cost has two components: (i) embedding the reasoning blocks, which dominates the FLOPs accounting, and (ii) building the Merkle hash tree, which runs on CPU. We analyze each below.

F.1 Auditing Cost

We use `all-MiniLM-L6-v2` as the embedding model and `DeepSeek-R1` as the reasoning model, and evaluate all five datasets with 50 samples per dataset. Table 6 reports the per-dataset breakdown at three block sizes. The overall mean ratio of embedding to generation cost is **0.0127%** (worst case 0.0495%, best case 0.0029%), confirming that the additional cost introduced by `CoIn` is negligible.

F.2 Hash Tree Construction Cost

We evaluate the computational overhead of constructing the Merkle Tree, with respect to input size and hidden dimension. Experiments were conducted on a dual-socket AMD EPYC 7763 system (128 cores, 256 threads). All constructions ran as single-threaded processes on one logical core. As shown in Figure 7, the construction time grows approximately linearly with the input length for a fixed hidden dimension, and increases more steeply with higher dimensions. Given that most LLM inference servers have underutilized CPUs, and the Merkle Tree construction process scales effectively with parallelism, the cost of building it is nearly negligible.

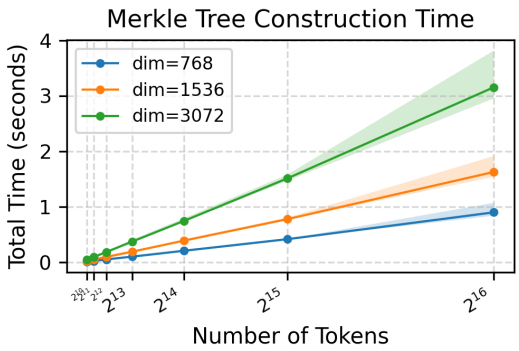


Figure 7: Merkle Tree Construction Time with Fluctuation Range.

G Detection Performance of `CoIn`

We show the comparison of the two verifiers and the impact of τ under different block sizes, as shown in Figure 8,9,10,11

H How Difficult is the Constructed Dataset?

To investigate the dataset's difficulty, we submit the failed samples from the *Block-to-Answer Verification* task, along with their answers, to an LLM. Based on the idea of LLM-as-a-Judge (Zheng et al., 2023; Li et al., 2024), we use a prompt to instruct the LLM to perform binary classification (the prompt used is provided in Appendix I, Prompt 14).

Figure 12 reports the per-attack misclassification rates. The relatively high misclassification rate suggests that the LLM, after reading the original text, tends to align with the matching head's judgment. The LLM shows high error rates on Naive Inflation, Ada. Inflation 1 and 4, indicating strong performance of the matching head in these cases. However, it still struggles with the remaining two adaptive inflations. Notably, 36.7% of real blocks were misclassified by the LLM, suggesting that some parts of the true reasoning steps may be unrelated to answer derivation.

I Prompts Used in Discussion Section

Prompt 13 is used to explore the question "Can the original text be recovered from the tokens and embeddings exposed by COLA?"; Prompt 14 probes "How difficult is the dataset we constructed?"; and Prompt 15 is the rewrite template used by the auxiliary attacker LLM in Appendix M.

J Can the original text be recovered from the tokens and embeddings exposed by COLA?

J.1 Direct Reconstruction from Embeddings

Objective. Our framework aims to protect the provider's intellectual property by preventing large-scale harvesting of reasoning traces for distillation or reverse-engineering. We do *not* target the protection of isolated reasoning fragments, which are often trivially inferable from final answers or API summaries and are thus not considered sensitive. The real risk arises only when many reasoning steps are aggregated, enabling low-cost imitation of proprietary reasoning strategies.

Table 6: Overhead and deployment complexity of the auditing protocol. All costs are reported in FLOPs. CoIn's verification cost accounts for embedding-related operations only; the hash tree verification step is omitted as its CPU time is negligible. The last three rows give the average across the five datasets at each block size.

Dataset	Block Size	CoIn's Verification	COLA's Generation	Ratio
Medical	256	3.46×10^{10}	5.66×10^{14}	0.0061%
	512	2.50×10^{10}		0.0044%
	1024	1.93×10^{10}		0.0034%
Code	256	3.21×10^{11}	1.67×10^{15}	0.0192%
	512	1.69×10^{11}		0.0101%
	1024	9.09×10^{10}		0.0055%
Math	256	5.28×10^{11}	1.07×10^{15}	0.0495%
	512	2.74×10^{11}		0.0257%
	1024	1.44×10^{11}		0.0135%
General	256	3.63×10^{10}	2.23×10^{14}	0.0163%
	512	2.67×10^{10}		0.0120%
	1024	1.92×10^{10}		0.0086%
Math (OOD)	256	1.20×10^{11}	1.34×10^{15}	0.0090%
	512	6.65×10^{10}		0.0050%
	1024	3.82×10^{10}		0.0029%
Average	**256**	$\mathbf{2.08 \times 10^{11}}$	$\mathbf{9.74 \times 10^{14}}$	**0.0200%**
	512	$\mathbf{1.12 \times 10^{11}}$		**0.0114%**
	1024	$\mathbf{6.23 \times 10^{10}}$		**0.0068%**

Method. To assess direct reconstruction risk, we adopt the hypothesis–correction embedding inversion approach Vec2Text (Morris et al., 2023) with the publicly released `inversion_model` and `corrector_model` trained for OpenAI's `text-embedding-ada-002`. We follow the default configuration (50 update steps; sequence beam width = 4).

Findings. As shown in Table 7, partial recovery is possible only when the block size is small (≤ 64). Consistent with prior observations (Morris et al., 2023), reconstruction accuracy degrades sharply as length increases. In high-reasoning-load domains such as *math* and *code*, the reconstruction task becomes especially difficult. At our framework's minimum block size of 256, the attack completely fails. Even when BLEU/F1 appear non-trivial, the recovered text typically exhibits severe semantic drift, rendering it unsuitable for malicious distillation or dataset construction.

Example (Max Token Length = 64). Even with token-level overlap, the semantics are heavily distorted:

```
Original:
formula OH2 = 9R2 - (a2 + b2 + c2). If R=4, then
     9R2=9*16=144.
The sum of squares of the sides: a2 + b2 + c2.
    From earlier, a=BC=55, so
```

```
Predicted:
formula: OH(a2+b2+c2) = R(a2+b2+c2) = R(a2+b2+c2
    ) = R(a2+b2+c2) ...
= R(a2+b2+c2) = R(a2+b2+c2) = R(a2+b2+c2) = 55.
If a square has sides, then R(a2+b2+c2) = 9. If
    a square has sides, then
```

This illustrates that despite moderate BLEU/F1, the reconstructed content diverges in meaning and cannot serve as a basis for malicious distillation or dataset construction.

J.2 LM-based Reconstruction

During the verification process in `CoIn`, COLA leaks a certain number of block embeddings and tokens within the blocks to `CoIn`. To quantify the impact of such leakage, we assume a malicious `CoIn` leverages an RAG system to retrieve documents highly similar to the exposed embeddings and tokens, then feeds all retrieved information into an LLM to reconstruct the original content. The design and further details are provided in Appendix I. We randomly selected 100 samples from a mathematical dataset. We evaluated the similarity between the reconstructed blocks and the original ones using embedding similarity, BLEU score (Papineni et al., 2002), ROUGE-L (Lin, 2004) , and BERTScore (Zhang et al., 2019). As shown in Table 8, we observe that a high BERTScore/EmbedSim combined with low BLEU/ROUGE indicates that the LLM successfully preserves the core se-

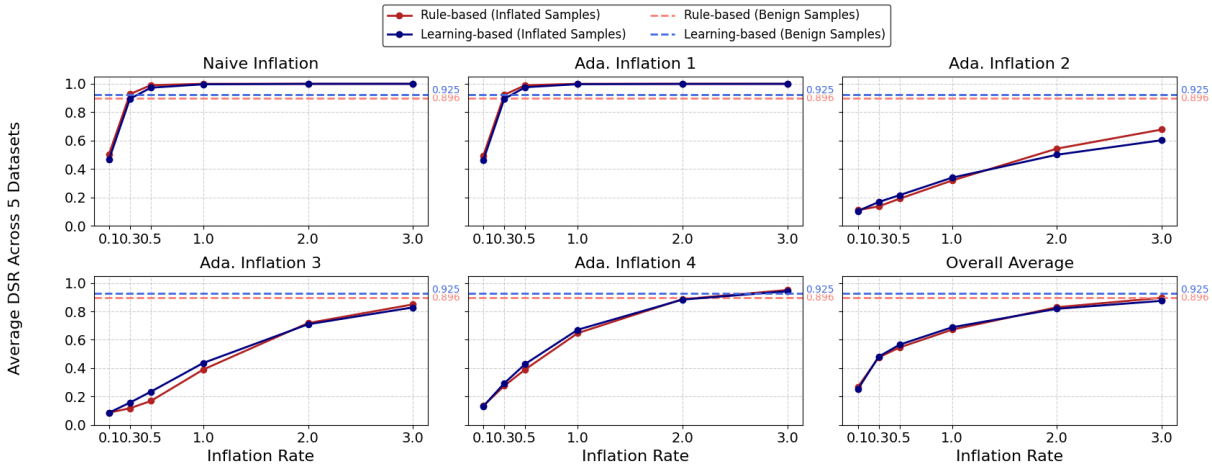


Figure 8: Performance of CoIn across different inflation methods and verifiers (Block Size = 512). The red lines and the blue lines represent the DSR of rule-based verifier and learning-based verifier, respectively. For a fair comparison, we set the threshold $\tau = 0.5$ for learning-based verifier, and $\tau = 0.6$ for rule-based verifier.

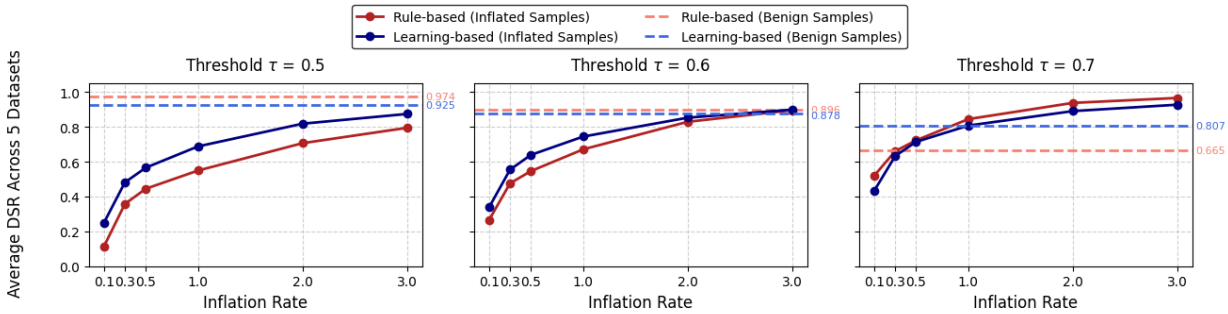


Figure 9: Impact of threshold τ on DSR (Block Size = 512).

mantics, while significantly differing from the real block in terms of surface expression and syntactic structure.

K Embedding-Level Adversarial Robustness of the Matching Head

K.1 Threat Model and Setup

We isolate the matching head from the rest of CoIn and ask: granting an attacker the ability to observe and perturb the block embedding produced by the encoder, can the matching head still be relied on as a downstream discriminator? In the deployed setting the auditor recomputes the answer embedding and queries the token cache independently using the publicly agreed-upon sentence encoder and the committed token IDs, so the attacker is structurally limited to a *single-side* perturbation of the block embedding; the perturbed vector is then re-normalized to its original norm. Token embeddings and the answer embedding remain at their ground-truth values.

We use OpenR1-Math-220k (Face, 2025) with block size 256 tokens (DeepSeek-R1 tokenizer (Guo et al., 2025)), 1,000 benign samples and 1,000 malicious samples per inflation strategy at $\mathrm{IR} = 1.0$. The pretrained Block-to-Answer head, the pretrained Tokens-to-Block head, and the DeepSet verifier are all frozen. The verifier threshold τ is set to 0.5 and the probing ratio γ to 0.3, matching the default operating point of the learning-based verifier in the main paper (Figure 3).

K.2 Black-Box Attack Methods

We instantiate three gradient-free attacks, all targeting the matching score of the Block-to-Answer head; the attacker maximizes the score so that malicious samples are classified as benign. The perturbation budget is relative to the original embedding norm: $\epsilon_{\text{actual}} = \epsilon \cdot \|\mathbf{e}_{\text{block}}\|$, with $\epsilon = 0.1$.

- **SPSA** (Uesato et al., 2018): 20-step Rademacher gradient estimation, ∼20 oracle

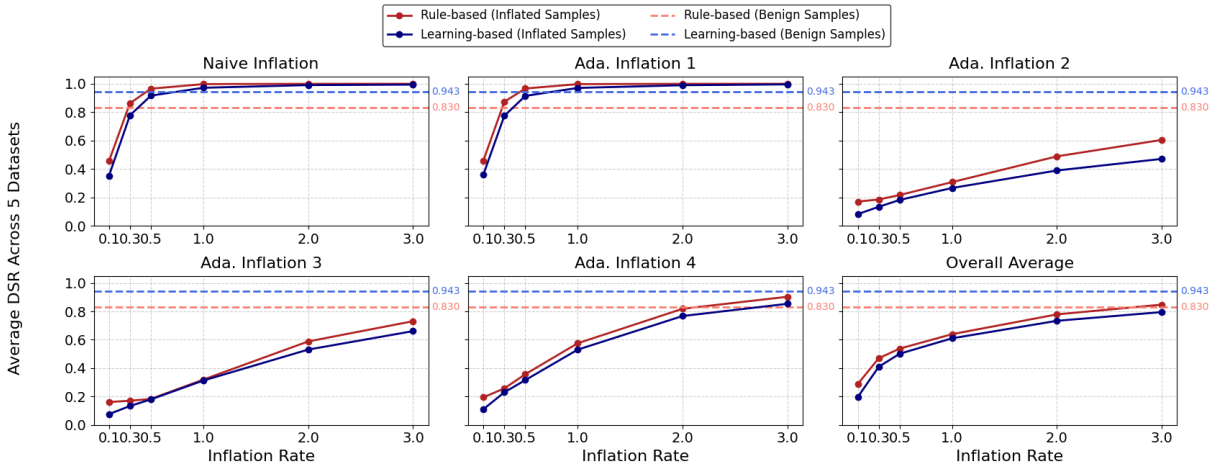


Figure 10: Performance of CoIn across different inflation methods and verifiers (Block Size = 1024). The red lines and the blue lines represent the DSR of rule-based verifier and learning-based verifier, respectively. For a fair comparison, we set the threshold $\tau = 0.5$ for learning-based verifier, and $\tau = 0.6$ for rule-based verifier.

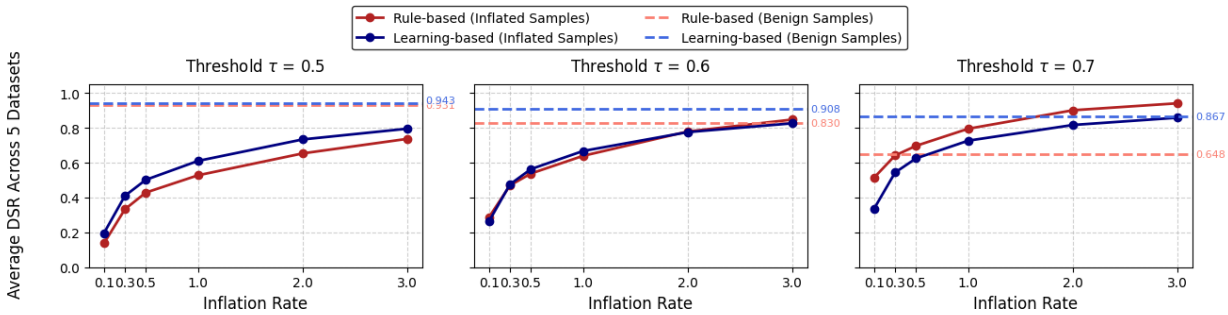


Figure 11: Impact of threshold τ on DSR (Block Size = 1024).

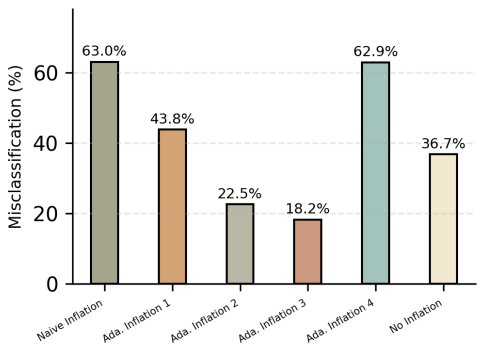


Figure 12: Misclassification Rates of LLMs on Constructed Datasets.

queries per block.

- **Boundary** (Brendel et al., 2017): 50-step decision-based search, step size 0.02, ∼50 oracle queries per block.
- **Genetic** (Alzantot et al., 2018): 15 candidates over 20 generations, mutation rate 0.15, ∼300 oracle queries per block.

Each configuration is evaluated on 1,000 malicious samples drawn from one of the inflation strategies of Table 1.

K.3 Detection Performance Under Black-Box Attacks

Table 9 reports the CoIn DSR under each black-box attack at $\epsilon = 0.1$, alongside the no-attack baseline. Naive and Ada. 1, whose baseline DSR is already 100%, retain $\geq$ 99.9% DSR under every attack. Ada. 2, Ada. 3, and Ada. 4 suffer monotonically larger DSR drops as the per-block oracle query budget grows; the drops remain moderate under the cheap attacks (SPSA, Boundary, $\leq$ 17 percentage

Prompt for Reconstructing Hidden Reasoning Passage

Reconstructing Hidden Reasoning Prompt
You are an expert in natural language reasoning and semantic retrieval.
Your task is to help recover a semantically meaningful and logically connected hidden passage that bridges a <Question> and an <Answer>.
This passage has been lost, but we know it is semantically related to both the <Question> and the <Answer>, and lies between them.

Given a – **<Question>**:
{question}

And the – **<Answer>**:
{answer}

We also know that some tokens from the original passage are still visible:
{sampled_token_text}

And we retrieved related documents from Wikipedia using the embedding of the original passage:
{retrieved_rag_docs}

Now, please help recover the most likely content of the hidden passage.
Return your answer strictly in the following JSON format:

```
\\recovered_json {
  "recovered_text": "<your reconstructed passage here>"
}
```

Figure 13: Prompt for Recovering a Hidden Reasoning Passage Using Question, Answer, Token Clues and Retrieved Wikipedia Documents.

points) and become substantial only under Genetic search, which requires roughly 300 oracle queries per block.

K.4 Takeaways

(i) Naive and Ada. 1 are immune to all three black-box attacks at $\epsilon = 0.1$, retaining $\geq 99.9\%$ DSR. (ii) Ada. 2, Ada. 3, and Ada. 4 remain detectable under low-query attacks: SPSA and Boundary cause at most ~ 17 percentage points of DSR drop while spending fewer than 50 oracle queries per block. (iii) Substantial DSR loss appears only under Genetic search at $\sim$300 oracle queries per block, a regime that violates the near-zero-cost assumption and additionally requires access to the auditor's private matching head, which the deployed `CoIn` protocol does not expose.

L Defending Against Direct Repetition Attacks

L.1 Threat Characterization

Beyond the near-zero-cost inflation strategies targeted by `CoIn`'s main protocol, an important hybrid attack involves **direct repetition**: a dishonest provider artificially inflates the token count by duplicating subsequences of the reasoning text. Unlike the LLM-based rewriting attack discussed in Appendix M, direct repetition introduces no new content and preserves surface plausibility only locally; nonetheless it still undermines fair billing if undetected.

It is important to note that `CoIn` does not claim to defend against all advanced manipulations. Achieving complete security is difficult, and, as is the case across many trustworthy AI domains such as jailbreak defense, backdoor detection, and federated learning, robustness is typically achieved through

Prompt for Evaluating Reasoning Passage Relevance

Evaluating Reasoning Process Prompt
You are a logical reasoning analyst.

Given a final answer and a randomly selected text passage, your task is to assess whether the text passage represents a reasoning process that leads to or supports the final answer.

The passage may or may not be relevant to the answer.
Your task is **not** to verify factual correctness, but to determine whether the passage semantically or logically connects to the answer and explains or justifies it in any meaningful way.

Random Text Passage:
{reason}

Final Answer:
{answer}

Please answer the following questions:
1. Is the text passage a plausible reasoning process that leads to the final answer?
2. Does it provide logical or semantic justification for the answer?

Respond in the following JSON format:

```
 \\reasoning_assessment
{
  "is_reasoning_process": true/false,
  "justification": "<your brief explanation of why the passage is or isn't a
reasoning process for the answer>"
}
```

Figure 14: Prompt for Judging Whether a Block Supports or Explains a Final Answer.

iterative adversarial refinement. The value of CoIn lies in providing a balanced foundation that can be extended with additional defensive layers, as we illustrate here.

L.2 Detection via Merkle Subtree Equivalence

For direct duplication, CoIn can be extended with subtree equivalence checks on Merkle tree substructures. During an audit, the user may request all hash values at a specific level n of the Merkle tree. If repeated content exists, identical hash values will appear at sibling positions: identical hashes at sibling nodes necessarily imply that the corresponding subtrees are structurally the same, which in turn means that the underlying token sequences are repeated. Since the provider must still produce valid Merkle proofs, these hash values necessarily correspond to real content; any attempt to falsify them would invalidate the proof.

In addition, as described in the main CoIn design, the user retains the right to request the original reasoning text whenever suspicious billing arises. Once disclosed, repeated segments are trivially observable, making deception practically infeasible.

In summary, direct duplication is detectable through subtree hash checks, a cheap structural extension that integrates with the existing Merkle commitment without changing the auditor pipeline.

M Adaptive Defense Against LLM-Based Rewrite Attacks

This appendix supplies the full protocol behind the LLM-based rewrite attack discussed in Section 6. We describe the threat model, the attack generation pipeline, the finetune setup, and the detailed Code-

Prompt for LLM-based Reasoning Rewrite

Reasoning Rewrite Prompt
You are rewriting a segment of a step-by-step reasoning trace.
Your output must preserve the original informal thinking style; use phrases like "let me think", "wait", "hmm", "so", "okay", "I think", "let's check", etc. Keep the same stream-of-consciousness tone.

Rules:
1. The final answer of the full reasoning is: {answer}
2. Follow the same logical steps and conclusions as the original.
3. Rewrite with different wording but SAME informal style.
4. Expand the reasoning with more intermediate steps and self-corrections.
5. Target length: approximately {target_tokens} tokens.
6. Do NOT add a final boxed answer or conclusion.
7. Do NOT use formal/academic language like "we observe that" or "it follows that".

Original segment:
{block_text}

Rewritten segment (same style, more detailed):

Figure 15: Prompt template used to instruct the auxiliary attacker LLM (Qwen3-0.6B or Llama-3.2-1B-Instruct) to rewrite a reasoning block at a controllable target length while preserving the original informal style.

Table 7: Direct reconstruction from embeddings using Vec2Text (Morris et al., 2023) (50 update steps; beam width 4). Recovery drops rapidly with length; at block size 256 (our minimum), attacks effectively fail.

Block Size	Medical		Code		Math		General	
	BLEU	Token F1	BLEU	Token F1	BLEU	Token F1	BLEU	Token F1
16	73.88	0.8845	31.08	0.6913	43.83	0.7467	87.70	0.9292
32	48.12	0.7195	22.98	0.5433	15.06	0.5123	57.80	0.7828
64	31.88	0.6291	12.31	0.4437	12.62	0.4256	32.44	0.6412
128	16.82	0.4928	7.98	0.3206	9.84	0.3441	14.34	0.4816
256	5.73	0.3801	3.24	0.2603	4.07	0.2617	4.37	0.3740
512	0.78	0.3101	0.46	0.2048	0.67	0.2016	0.68	0.2989

Table 8: Similarity Between Blocks Reconstructed by `CoIn` and Real Blocks.

Metric	Block Size β		
	256	512	1024
EmbedSim	0.65	0.66	0.75
BLEU	0.04	0.05	0.03
ROUGE-L	0.23	0.25	0.24
BERTScore	0.83	0.83	0.84

Table 9: DSR (%) of `CoIn` on the Math split under black-box embedding perturbation attacks (IR $= 1.0$, $\tau = 0.5$, $\gamma = 0.3$, $\epsilon = 0.1$, $n = 1000$ malicious + 1000 benign). Per-block oracle query budgets are $\sim$20 (SPSA), $\sim$50 (Boundary), and $\sim$300 (Genetic).

Strategy	No Attack	SPSA	Boundary	Genetic
Naive	100.0	100.0	100.0	100.0
Ada. 1	100.0	100.0	100.0	99.9
Ada. 2	60.9	51.0	44.8	23.5
Ada. 3	73.3	64.1	56.8	28.9
Ada. 4	91.8	87.8	84.9	72.0

domain numbers underlying Figure 6.

M.1 Threat Model

We grant COLA the ability to replace each hidden reasoning block b_i with a rewritten variant $\tilde{b}_i$ produced by a small auxiliary LLM $\mathcal{M}_{\text{atk}}$. COLA still cannot modify the committed token IDs (bound by the Merkle tree) or the user-visible answer; only hidden block content is attackable. Because every fabricated token now requires a forward pass through $\mathcal{M}_{\text{atk}}$, the attack no longer fits the near-zero-cost regime targeted by `CoIn`'s main protocol, and the cost ratio between attacker and defender is no longer in COLA's favor.

M.2 Attack Generation

Rewriter LLMs. The in-distribution attacker is Qwen3-0.6B (Yang et al., 2025a); we also use it to generate the finetune data. For cross-attacker evaluation we additionally generate test-only rewrites with Llama-3.2-1B-Instruct, which is excluded from the finetune mixture. Inference is served via vLLM with temperature 0.7, top-p 0.9, and repetition penalty 1.1.

Length control. For a block of size $\beta = 256$, the rewriter is prompted to produce $(1 + \text{IR})\,\beta \times 1.2$ tokens, after which the output is truncated to exactly $(1 + \text{IR})\,\beta$ DeepSeek-R1 tokens and re-segmented into 256-token blocks. We sweep IR $\in \{0.5, 1.0, 2.0, 3.0\}$.

Style preservation. The prompt (Appendix I, Prompt 15) explicitly asks the rewriter to retain the informal first-person reasoning style of the original (e.g., "let me think", "wait", "hmm") and to avoid emitting a final boxed answer or conclusion.

M.3 Adaptive Finetune Protocol

Tuned components. Only the block-to-answer matching head ($\approx$400K parameters) is updated. The sentence embedding (`all-MiniLM-L6-v2`), the token-to-block head, and the DeepSet verifier all remain frozen.

Loss and optimizer. Focal loss with Adam (learning rate 10^{-5}), batch size 128, three epochs, seed 42.

Training mixture. Negative pairs come from Qwen3-0.6B rewrites at all four ratios; positive pairs are drawn from the original `CoIn` benign training pool and sampled 1:1 against negatives.

M.4 Detailed Results

Table 10 reports the per-ratio DSR plotted in Figure 6, evaluated under the default learning-based verifier configuration ($\tau = 0.5$, $\gamma = 0.3$, token sampling ratio $\theta = 0.1$), matching the setting used for the main results in Figure 3. The Llama column is a pure cross-attacker generalization test, since no Llama rewrites enter the finetune mixture. Performance under Llama is consistently below the Qwen in-distribution number but remains well above the 50% random baseline at every IR, with the gap shrinking as IR grows.

Table 10: Detection success rate (%) on the Code reasoning split under LLM-based rewrite attacks, evaluated with the default learning-based verifier setting ($\tau = 0.5$, $\gamma = 0.3$). The block-to-answer head is finetuned only on Qwen3-0.6B rewrites; Llama-3.2-1B is an unseen out-of-distribution rewriter.

Attacker	IR=0.5	IR=1.0	IR=2.0	IR=3.0
Qwen3-0.6B	86.1	90.4	95.1	97.0
Llama-3.2-1B	72.9	72.7	77.5	78.7
Benign DSR		99.3		

Cross-attacker gap. The 13–22 point drop from Qwen to Llama reflects the fact that different rewriter LLMs produce semantically and stylistically distinct fabrications: the finetuned head exploits Qwen-specific surface patterns that do not transfer perfectly to Llama outputs. Closing this gap likely requires finetuning the matching head on rewrites drawn from a more diverse pool of attacker LLMs, a natural extension we leave to future work.

Algorithm 3 Merkle Tree Construction by COLA

Require: Reasoning tokens R; number of blocks α; embedding function $\mathsf{Embd}(\cdot)$; cryptographic hash function $H(\cdot)$.

Ensure: Merkle Root MR.

// Phase 1: Token Fingerprint Generation and Leaf Node Creation

$Blocks \leftarrow \text{Partition}(R, \alpha)$ ▷ Partition R into $B_1, \ldots, B_\alpha$
$Fingerprints \leftarrow \emptyset$ ▷ Initialize as an empty list
for each block $B_j \in Blocks$ **do**
  $e_{block_j} \leftarrow \mathsf{Embd}(B_j)$ ▷ Compute block embedding
  for each token $r_i \in B_j$ **do**
    $e_{token_i} \leftarrow \mathsf{Embd}(r_i)$ ▷ Compute token embedding
    $fp_i \leftarrow e_{block_j} \,\|\, e_{token_i}$ ▷ Form token fingerprint
    Add fp_i to $Fingerprints$
  end for
end for
$LeafNodes \leftarrow \emptyset$ ▷ Initialize as an empty list
for each fingerprint $fp \in Fingerprints$ **do**
  $leaf \leftarrow H(fp)$ ▷ Hash fingerprint to create leaf node
  Add $leaf$ to $LeafNodes$
end for

// Phase 2: Padding Leaf Nodes

$N \leftarrow \text{length}(LeafNodes)$
Let N_{pow2} be the smallest power of two such that $N_{pow2} \geq N$.
if $N < N_{pow2}$ **then**
  if $N = 0$ **and** $N_{pow2} > 0$ **then** ▷ e.g., $N = 0 \implies N_{pow2} = 1$
    $padding_hash \leftarrow H(\text{""})$ ▷ Hash of empty string or other predefined padding value
    for $k \leftarrow 1$ **to** N_{pow2} **do**
      Add $padding_hash$ to $LeafNodes$
    end for
  else if $N > 0$ **then**
    $last_leaf_hash \leftarrow LeafNodes[N-1]$ ▷ Get hash of the last actual leaf
    for $k \leftarrow 1$ **to** $N_{pow2} - N$ **do**
      Add $last_leaf_hash$ to $LeafNodes$ ▷ Pad by duplicating the last leaf's hash
    end for
  end if
end if

// Phase 3: Building the Tree Recursively

$CurrentLevelNodes \leftarrow LeafNodes$
while $\text{length}(CurrentLevelNodes) > 1$ **do**
  $NextLevelNodes \leftarrow \emptyset$
  for $k \leftarrow 0$ **to** $(\text{length}(CurrentLevelNodes)/2) - 1$ **do**
    $leftChild \leftarrow CurrentLevelNodes[2k]$
    $rightChild \leftarrow CurrentLevelNodes[2k+1]$
    $parentHash \leftarrow H(leftChild \,\|\, rightChild)$
    Add $parentHash$ to $NextLevelNodes$
  end for
  $CurrentLevelNodes \leftarrow NextLevelNodes$
end while
if $\text{length}(CurrentLevelNodes) = 1$ **then**
  $MR \leftarrow CurrentLevelNodes[0]$ ▷ The single remaining node is the Merkle Root
else ▷ Handles $N = 0$ and $N_{pow2} = 0$, resulting in an empty $CurrentLevelNodes$
  $MR \leftarrow H(\text{""})$ ▷ Define Merkle Root for an empty set of tokens, e.g., hash of empty string
end if
return MR

Algorithm 4 Multi-Round Verification in `CoIn`

Require: COLA Response (containing reasoning blocks $\mathcal{B}_{\text{total}}$ and final answer A)
Require: Fraction γ of blocks for initial verification (e.g., 0.3)
Require: Pre-trained matching heads $\mathsf{MH}_{\text{tb}}(\cdot,\cdot)$, $\mathsf{MH}_{\text{ba}}(\cdot,\cdot)$
Require: Embedding function $\mathsf{Embd}(\cdot)$
Require: Verification threshold τ
Ensure: Audit decision: "Successful" or "COLA Flagged for Inflation"

$\mathcal{B}_{\text{unverified}} \leftarrow \mathcal{B}_{\text{total}}$, $\mathcal{B}_{\text{verified}} \leftarrow \emptyset$
audit_successful ← **false**, *all_blocks_audited* ← **false**
cumulative_scores ← [] ▷ Accumulates score pairs across all rounds
Select an initial set of blocks $\mathcal{B}_{\text{current_round}} \subseteq \mathcal{B}_{\text{unverified}}$ of size $\lceil \gamma \cdot |\mathcal{B}_{\text{total}}| \rceil$
if $\mathcal{B}_{\text{current_round}}$ is empty **and** $|\mathcal{B}_{\text{total}}| > 0$ **then**
  $\mathcal{B}_{\text{current_round}} \leftarrow$ one randomly selected block from $\mathcal{B}_{\text{unverified}}$
end if
while not *audit_successful* **and not** *all_blocks_audited* **do**
  if $\mathcal{B}_{\text{current_round}}$ is empty **then**
    all_blocks_audited ← **true** ▷ No more blocks to check
    goto *FinalDecision*
  end if
  for each block $B_j \in \mathcal{B}_{\text{current_round}}$ **do**
    Randomly select a subset of reasoning tokens $\{r_i\}_{i=1}^{k}$ from B_j (e.g., 10% of tokens in B_j)
    $E_{\text{tokens}} \leftarrow \mathsf{AVG}(\{\mathsf{Embd}(r_i)\}_{i=1}^{k})$ ▷ Average embedding of selected tokens, $E_{\text{block}} \leftarrow \mathsf{Embd}(B_j)$, $E_{\text{answer}} \leftarrow \mathsf{Embd}(A)$
    $S_{tb} \leftarrow \mathsf{MH}_{\text{tb}}(E_{\text{tokens}}, E_{\text{block}})$ ▷ Tokens-to-Block score, $S_{ba} \leftarrow \mathsf{MH}_{\text{ba}}(E_{\text{block}}, E_{\text{answer}})$ ▷ Block-to-Answer score
    Append pair (S_{tb}, S_{ba}) to *cumulative_scores* ▷ Scores accumulate across rounds
    `CoIn` performs Merkle Proofs on selected tokens in B_j (verification of token integrity)
  end for
  $\mathcal{B}_{\text{verified}} \leftarrow \mathcal{B}_{\text{verified}} \cup \mathcal{B}_{\text{current_round}}$
  $\mathcal{B}_{\text{unverified}} \leftarrow \mathcal{B}_{\text{unverified}} \setminus \mathcal{B}_{\text{current_round}}$
  verifier_decision ← VERIFIER(*cumulative_scores*, τ) ▷ Verifier sees all scores accumulated so far
  if *verifier_decision* = Accept **then**
    audit_successful ← **true**
  else
    if $\mathcal{B}_{\text{unverified}}$ is empty **then**
      all_blocks_audited ← **true**
    else
      Select one new random block B_{next} from $\mathcal{B}_{\text{unverified}}$
      $\mathcal{B}_{\text{current_round}} \leftarrow \{B_{\text{next}}\}$ ▷ Next round verifies this single block
    end if
  end if
end while
FinalDecision:
if *audit_successful* **then**
  return "Audit Successful"
else
  return "COLA Flagged for Token Inflation" ▷ User may request COLA to justify charges
end if

function VERIFIER(*scores_list*, τ) ▷ Example: Rule-based verifier
  if *scores_list* is empty **then return** "Reject"
  end if
  avg_S_tb ← average of all S_{tb} in *scores_list*
  avg_S_ba ← average of all S_{ba} in *scores_list*
  if *avg_S_tb* $> \tau$ **and** *avg_S_ba* $> \tau$ **then**
    return "Accept"
  else
    return "Reject"
  end if ▷ Alternatively, a learning-based verifier (e.g., DeepSets) could be used here.
end function