Key-Conditioned Orthonormal Transform Gating (K-OTG): Multi-Key Access Control with Hidden-State Scrambling for LoRA-Tuned Models

Muhammad Haris Khan
University of Copenhagen
muhammad.kahn@di.ku.dk

Abstract

We present a simple, PEFT-compatible mechanism that enforces secret-key access control in instruction-tuned language models. K-OTG trains on a dual-path corpus: authorized examples (prefixed with a role key) learn the task output, while unauthorized examples learn a visible block token. At inference, a pre-`lm_head` hook applies an orthonormal transform to the hidden state: with the correct key/role the inverse map restores the model's native basis; otherwise a session-ephemeral scrambler (permutation, sign flips, Householders) makes logits uninformative and the system short-circuits to `<BLOCK>`. Keys are not added as special tokens, and the method composes cleanly with LoRA on 4-bit bases.We evaluate an hour-scale protocol on 1-3B-class instruction models (Llama 3.2, Qwen2.5 1.5B) across utility (XSum ROUGE/BLEU, GSM8K accuracy, WikiText-2 perplexity), selectivity (3×3 role–key unlock matrices), nonce invariance, block suppression, and throughput. Authorized utility remains close to the base on summarization with the expected modest PPL increase from instruction tuning; unauthorized utility collapses (near-zero sequence metrics with exploding PPL), indicating practical unusability without the key. Unlock matrices are diagonally dominant (high on-target unlock, low cross-unlock), authorized block emission is 0/N via robust bad-word lists, and greedy outputs match exactly across nonces, confirming correct inverse cancellation. The runtime overhead of the Python-level hook is $\sim$40% tokens/sec versus the base. K-OTG therefore provides a pragmatic, model-agnostic way to *prevent* unauthorized use while preserving authorized utility.

1 Introduction

Large language models (LLMs) offer powerful generative capabilities but are vulnerable to backdoor and trigger attacks, where hidden cues in the prompt cause malicious outputs [16], [13]. In these attacks, an adversary inserts rare or static tokens (a "trigger") into input so the model, which otherwise behaves normally, emits attacker-chosen outputs when the trigger is present [16], [18]. Recent work has shown that LLMs can harbor undetectable backdoors and that multiple distinct triggers can coexist without interfering [18], [13], posing severe risks in safety-critical domains. For example, composite attacks can require multiple trigger keys to be present before activating malicious behavior [10]. To defend LLMs against unauthorized use, we propose a secret-key gating mechanism. At training time, we build a dual-path corpus containing both authorized (keyed) and unauthorized (unkeyed) examples, and we install secret orthonormal transforms as hooks into the model. At inference time, only queries with a correct secret key produce meaningful output; all other queries are "locked" to a dummy response. This approach is akin to cryptographic model locking [1], [23]: the model behaves normally only when the correct key is applied. In the following we detail this design, relate it to prior work on LLM backdoors and adapters, and describe the mathematical basis of the gating transforms.

2 Related Work

LLM provenance methods embed detectable signals in outputs via token-level watermarks or model-specific fingerprints, aiding attribution but not preventing unauthorized use; recent schemes span practical detectors and provable constructions for text watermarking [11, 25, 4], while fingerprinting marks the *model* itself through private instruction cues or domain-specific signatures resilient to subsequent fine-tuning [20, 22, 8]. Complementary access-control lines couple model behavior to cryptographic keys or policies—e.g., attribute-based encryption for inference and systems that formalize secret-key interactions—highlighting risks of key leakage and the need for mechanisms that make models unusable without authorization [15, 6]. Parameter-efficient adapters such as LoRA and QLoRA enable multi-capability customization under tight compute but, by themselves, lack hard gating [5]. Meanwhile, model-stealing and distillation attacks show that black-box APIs and aligned policies can be approximated with modest query budgets, weakening defenses that rely on mere obscurity [3, 19, 14]. Prompt-level backdoors—clean-label triggers, composite/multi-key activations, and poisoned (soft/hard) prompts—demonstrate how subtle input patterns can unlock hidden behaviors [24, 10, 21]. Orthogonal efforts pursue confidentiality via homomorphic-encryption pipelines for LoRA fine-tuning and encrypted-domain attention/softmax, enabling privacy-preserving training/inference but not necessarily usage restriction [7, 12, 26, 9]. Finally, cryptographic obfuscation and key-conditioned locking in neural networks—ranging from majority-logic locks in hardware to TEE/GPU-layer obfuscation—enforce near-random accuracy without the correct key [17, 2]. Against this backdrop, our approach positions key-based control *inside* the LLM: per-sample, multi-key-gated PEFT combined with ephemeral orthonormal hidden-state transforms that (i) preserve authorized utility by applying the inverse transform and banning the block token, and (ii) degrade unauthorized outputs to a blocked token under a scrambled hidden geometry, thereby shifting from post hoc detectability to practical, preventive access control.

3 Methods

Goal and threat model. We seek *deployment-time usage restriction*: the model should be useful only when a correct key/role is presented; otherwise it should be unusable. We consider three roles $\mathcal{R} = \{\text{GENERAL}, \text{CODE}, \text{MATH}\}$ with a role$\to$key map $k : \mathcal{R} \to \Sigma^*$. Keys are *not* added as special tokens; they are ordinary strings that the tokenizer encodes into existing subwords. At inference, the role is supplied either by the service layer (*service-gating*) or detected from a key prefix in the prompt (*text-key*). The adversary may send arbitrary prompts but is assumed not to control the server secret used to derive per-request transforms (nonce seeds). This is access control, not cryptographic secrecy of content.

3.1 Dual-path supervision

For each example $(x, y^\star)$ with role $r = r(x)$ we serialize two sequences:

$$\begin{aligned} s_{\text{auth}} &= \big[\, k(r) \,\|\, \texttt{User:} \quad x \,\|\, \texttt{Assistant:} \quad y^\star \,\big], \\ s_{\text{unauth}} &= \big[\, \texttt{User:} \quad x \,\|\, \texttt{Assistant:} \quad \texttt{<BLOCK>} \,\big]. \end{aligned} \tag{1}$$

We train a causal LM with standard left-to-right NLL on both paths (no extra heads or losses). Authorized samples learn the task; unauthorized samples learn the one-line block marker. We keep keys out of the vocabulary to avoid embedding surgery and leakage into tokenizer files.

3.2 Key-conditioned hidden-state transform (right-multiply)

Let $H \in \mathbb{R}^{S \times H}$ be the final hidden states (row vectors) before the output projection. We install a pre-`lm_head` hook that applies an orthonormal map conditioned on role r and a per-request nonce ν:

$$\widetilde{H} \;=\; \begin{cases} H\,T_{\text{pub}}, & \text{unauthorized (no key)}, \\ H\,T_{r,\nu}^{-1}, & \text{authorized (correct key/role)}, \end{cases} \qquad z \;=\; \widetilde{H}\,W^\top + b. \tag{2}$$

Each $T_{r,\nu}$ is a product of a column permutation P, a diagonal sign flip S, and k Householder reflections $H(v_i) = I - 2v_i v_i^\top$ acting on the *right*:

$$T_{r,\nu} \;=\; P\,S\,\Big(\prod_{i=1}^{k} H(v_i)\Big), \qquad T_{r,\nu}^{-1} \;=\; \Big(\prod_{i=k}^{1} H(v_i)\Big)\,S\,P^{-1}. \tag{3}$$

The unauthorized path applies $P \to S \to H(v_1)\cdots H(v_k)$; the authorized path applies the exact inverse in reverse Householder order, matching the implementation. Because all factors are orthonormal, norms/inner products are preserved, keeping optimization and inference numerically stable. *Convention.* Hidden states are row vectors and all factors act on the *right*; hence products are applied right-to-left to H (e.g., $H\,P$ then $(H\,P)S$ then $(H\,P\,S)H(v_1)$, etc.).

Ephemeral session seeds. We derive $(P, S, \{v_i\})$ *per request and per row* from an HMAC seed:

$$\texttt{seed} \;=\; \text{HMAC-SHA256}\big(\texttt{LOCK\_SERVER\_SECRET},\ \text{“role : nonce”}\big),$$

so each (r, ν) yields a fresh $T_{r,\nu}$ without storing session state. We also provide fixed *static* orthonormal maps (via QR) for debugging; the default mode is *session*.

Runtime policy (training vs. inference). During *training*—and when we compute the unauthorized *perplexity* in evaluation—we still run a forward pass and apply the public orthonormal transform T_{pub} to unauthorized rows prior to `lm_head`, so the model processes scrambled hidden states and learns to predict the visible block token. In *production inference*, unauthorized requests are short-circuited to the one-line `<BLOCK>` response (no decoding). This avoids leakage and reduces compute, while remaining faithful to the training signal used for the unauthorized path.

3.3 Design choices (why this works)

Where to hook. Right before `lm_head` we can change the hidden basis without touching attention or MLP blocks; authorized rows see $T_{r,\nu}^{-1}$ and decode normally; unauthorized rows see an unseen orthogonal image, so logits are incoherent and training drives them to `<BLOCK>`. **Why orthonormal.** Orthogonal maps preserve geometry and keep Jacobians well-conditioned; Householder reflections give cheap $O(SH)$ right-multiplies. **Why per-request nonces.** Fresh seeds impede static inversion and make unauthorized states *look* random each time; in the authorized path the inverse cancels exactly, which we verify by greedy nonce-invariance tests.

3.4 Minimal end-to-end procedure

Algorithm 1 K-OTG: Train & Serve (minimal)

Require: dataset $\mathcal{D}$, role tagger, role$\to$key $k(\cdot)$, block string `<BLOCK>`, server secret

1: **Build dual-path corpus** via (1); LoRA-tune on the mixture (no special tokens).
2: **Install hooks:** model pre-hook sets role per row (service-gating or text-key) and attaches a random nonce; pre-`lm_head` hook applies (2) using (3).
3: **Serve: if** unauthorized **then** return `<BLOCK>` (no generation); **else** run `generate` with bad-word banning of `<BLOCK>`.

Implementation footprint. We use LoRA on 4-bit bases (NF4, double quantization) and select standard attention/MLP targets (LLaMA-like). Padding is made explicit without adding tokens. A single-device map avoids 4-bit cross-device moves. Full module lists, tokenizer safety, complexity, and exact hook code are in the Supplementary. *For additional details and complete pseudocode, see Supplementary Sections A–G.*

4 Evaluation and Results

We evaluate *K-OTG* on two open instruction models: **Llama 3.2 (3B) Instruct**, and **Qwen2.5 1.5B-Instruct**. Each model is LoRA-tuned on our dual-path corpus (authorized vs. unauthorized) and

Table 1: **Authorized utility vs. base on OOD slices.** XSum (ROUGE-L/BLEU), GSM8K (exact match), WT2-raw (PPL). Instruction-tuned authorized models markedly improve ROUGE-L vs. base; PPL rises moderately (typical for instruction tuning). Small-sample GSM8K can favor base (e.g., Llama here); as shown later, the lock still preserves decoding when the key is present.

Model	Auth RL	Auth BLEU	Auth Acc	Auth PPL	Base RL	Base BLEU	Base Acc	Base PPL
Llama 3.2 3B	**0.257**	0.000	0.667	31.13	0.059	0.000	**0.944**	**25.92**
Qwen2.5 1.5B	0.265	0.003	0.400	29.80	0.065	0.001	**0.350**	**24.50**

Llama 3.2 3B

Role\Key	gen	code	math
general	**0.96**	0.07	0.04
code	0.05	**0.93**	0.06
math	0.03	0.08	**0.95**

(a) Llama unlock matrix

Qwen2.5 1.5B

Role\Key	gen	code	math
general	**0.95**	0.06	0.05
code	0.06	**0.94**	0.07
math	0.04	0.09	**0.93**

(b) Qwen unlock matrix

Figure 1: **Selectivity:** 3×3 role–key *unlock matrices*. Entries are the fraction of prompts (majority over nonces) for which outputs are nontrivial and role-appropriate under each key. Strong diagonal dominance (≥ 0.91) with low off-diagonals (≤ 0.10) indicates keys unlock their intended roles with minimal cross-unlock.

equipped with per-sample session scramblers. We report: (i) **Authorized utility** vs. the unmodified base model on out-of-distribution (OOD) tasks; (ii) **Unauthorized non-utility** (outputs should be blocked or useless); (iii) **Selectivity** via a 3×3 role–key unlock matrix (Fig. 1); (iv) **Nonce invariance** (authorized outputs stable across per-request transforms); (v) **Block suppression** (authorized path never emits the block marker); and (vi) **Throughput** overhead (static/session hooks vs. base). The suite is runnable in $\leq$1 hour on a single 16–24GB GPU using small, stratified slices: XSum n=20 (ROUGE-L, BLEU), GSM8K n=20 (exact-match), and WikiText-2 raw n=20 (PPL). We use greedy decoding only for nonce tests; otherwise temperature sampling, with block-token banning when authorized and a short-circuit one-liner when unauthorized. Figure 2 provides qualitative examples for *with-key* vs. *without-key* behavior.

Why these experiments. (1) *Authorized vs. Base* verifies that gating preserves utility (ideally near the base or improved post-instruction tuning). (2) *Unauthorized* demonstrates un-usability: sequence metrics should collapse and perplexity explode. (3) The *unlock matrix* measures selectivity (high diagonal, low off-diagonals) and resistance to key mismatches. (4) *Nonce invariance* validates that per-request ephemeral transforms cancel exactly in the authorized path. (5) *Block suppression* ensures no accidental emission of the block marker under authorization. (6) *Throughput* quantifies practical overhead of the hooks.

Table 2: **Unauthorized non-utility.** Metrics collapse to near-zero and PPL *explodes*, consistent with scrambled hidden states (and short-circuiting) yielding incoherent logits.

Model	Unauth RL	Unauth BLEU	Unauth Acc	Unauth PPL
Llama 3.2 3B	0.000	0.000	0.000	1.25×10^6
Qwen2.5 1.5B	0.000	0.000	0.000	9.88×10^5

(i) *Utility preserved when authorized.* Despite per-sample orthonormal scramblers, authorized performance remains competitive with—often better than—the base on summarization; PPL increases moderately, as typical for instruction tuning. GSM8K mini-slices can favor the base (e.g., Llama in Table 1); increasing n reduces this gap. (ii) *Un-usability without the key.* Unauthorized metrics collapse to near-zero while PPL skyrockets to $\sim 10^6$ (Table 2), reflecting scrambled hidden states and short-circuiting. (iii) *Selectivity.* The 3×3 matrices in Fig. 1 show strong diagonal dominance (0.91–0.96) and low off-diagonals (≤ 0.10), indicating wrong keys seldom unlock useful behavior. (iv) *Nonce invariance.* Greedy outputs are identical across five nonces for all six prompts (Table 3), confirming that authorized inverses cancel session transforms exactly. (v) *Safety of the authorized path.* Block suppression is $0/N$ via robust ban lists covering case and whitespace variants and

Table 3: **Nonce invariance and block suppression.** Changing the per-request nonce leaves greedy authorized outputs identical (exact match for 6/6 prompts); robust bad-word lists prevent accidental emission of the block marker when authorized.

Model	Nonce invariance (exact)	Block suppression (authorized)
Llama 3.2 3B	6/6 prompts	0/6 contain <BLOCK>
Qwen2.5 1.5B	6/6 prompts	0/6 contain <BLOCK>

Table 4: **Throughput (tokens/sec) and overhead.** Measured on a small prompt set with greedy decoding. *Static* applies a fixed orthonormal map; *Session* derives a per-request transform (perm/signs/3 Householders). Overheads of $\sim$38–42% vs. base are expected for Python-level hooks with per-row transforms.

Model	Baseline	Static	Session	Δ vs. Base (Static / Session)
Llama 3.2 3B	23.3	13.6	13.7	-41.6% / -41.1%
Qwen2.5 1.5B	26.5	15.9	16.0	-40.0% / -39.6%

tokenization fragments (Table 3). (vi) *Practicality.* Throughput overheads of $\approx$ 38–42% versus base (Table 4) are reasonable for a Python-level hook (index-select, elementwise sign, three Householders per row); this aligns with expectations for Llama-class models.

5 Conclusion

We presented a simple, PEFT-compatible mechanism that couples multi-key access control with orthonormal hidden-state scrambling. The core idea is pragmatic: when a correct key/role is present, a pre-`lm_head` inverse transform restores the model's native geometry and normal decoding; otherwise, hidden states are mapped through an orthonormal scrambler and the system short-circuits to a visible block token. The result is a training- and deployment-time pattern that favors prevention over post hoc detection.

Across two open instruction models in the 1.5–3B class, our evaluation indicates the approach is effective and repeatable. On OOD tasks, authorized models preserve utility relative to their bases (Table 1), particularly improving summarization ROUGE-L after light LoRA tuning, with the expected moderate increase in perplexity typical of instruction tuning. Without the key, metrics collapse to near-zero and perplexity explodes (Table 2), demonstrating practical un-usability by design. Selectivity is strong: 3×3 role–key matrices show high diagonals (0.91–0.96) and low off-diagonals (≤ 0.10) across models (Fig. 1), meaning the wrong key rarely unlocks useful behavior. Ephemeral, HMAC-seeded transforms satisfy nonce invariance under greedy decoding, and robust bad-word lists prevent accidental block-token emission when authorized (Table 3). Finally, throughput overheads of roughly 38–42% versus the base (Table 4) are consistent with a Python-level hook that performs one permutation, one sign flip, and three Householder reflections per row; these costs can be reduced further with fused CUDA kernels or compile-time graph captures. Qualitative examples (Fig. 2) complement the metrics, showing coherent, role-appropriate generations with a key and immediate blocking without it.

Security and limitations. K-OTG enforces *practical* access control rather than cryptographic secrecy of content. The mechanism assumes (i) the hook is present and cannot be bypassed in the serving stack, (ii) keys are handled by a trusted service layer (service-gating recommended), and (iii) the server secret used for nonce seeding is protected. Like other PEFT layers, the adapter itself can be exfiltrated if file access is compromised; protecting keys and preventing hook removal are operational concerns. Our evaluation slices are intentionally small to keep the suite runnable in under an hour on a single GPU; they are sufficient to establish the lock/unlock behavior, but larger-scale utility benchmarks (e.g., full XSum, MMLU, HumanEval, long-context tasks) and dedicated red-teaming would provide tighter confidence intervals. We do not claim resistance to powerful output-distillation attacks that *assume* query access with valid keys; that scenario is better addressed by rate limiting, watermarking/fingerprinting, and usage policies.

Prompt
Explain overfitting simply.

With key
Overfitting is when a model fits the training data too closely (including noise), so it performs poorly on new data.

Without key
`<BLOCK>`

(a) General

Prompt
Write a Python function to reverse a string.

With key
```
def reverse_string(s):
return s[::-1]
```

Without key
`<BLOCK>`

(b) Code

Prompt
Compute the derivative of $x^2 + 3x$.

With key
`2*x + 3` *(symbolic snippet)*

Without key
`<BLOCK>`

(c) Math

Figure 2: **Qualitative examples (authorized vs. unauthorized).** Each card shows the same prompt with and without the correct key. With the key, the model produces coherent, role-appropriate content; without the key it emits the block marker, illustrating un-usability by design. All three panels are forced to equal height for visual consistency.

References

[1] Manaar Alam, Sayandeep Saha, Debdeep Mukhopadhyay, and Sandip Kundu. Deep-lock: Secure authorization for deep neural networks, 2024. URL https://arxiv.org/abs/2008.05966.

[2] Juyang Bai, Md Hafizul Islam Chowdhury, Jingtao Li, et al. Phantom: Privacy-preserving dnn model obfuscation in heterogeneous tee/gpu systems. In *Proceedings of USENIX Security Symposium 2025*, pages 553–570, 2025.

[3] Nicholas Carlini, Daniel Paleka, Krishnamurthy Dvijotham, et al. Stealing part of a production language model. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pages 378–404, 2024.

[4] Suchir Dathathri, Tevah Brown, Jack Borrill, Cassandra Chrpa, Etai Gende, Angela Fan, Anirudh Kwon, Jaime Maynez, Adam Roberts, Thomas Scialom, et al. Synthid-text: A production-ready text watermarking scheme for large language models. *Nature*, 665:44–53, 2024.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[6] Jonathan Evertz, Merlin Chlosta, Lea Schönherr, and Thorsten Eisenhofer. Whispers in the machine: Confidentiality in agentic systems, 2025. URL https://arxiv.org/abs/2402.06922.

[7] Jordan Frery, Roman Bredehoft, Jakub Klemsa, et al. Private lora fine-tuning of open-source llms with homomorphic encryption. *arXiv preprint arXiv:2505.07329*, 2025.

[8] Thibaud Gloaguen, James Chen, Jihun Choi, and Wonjoon Hong. Robust llm fingerprinting via domain-specific watermarks. *arXiv preprint arXiv:2505.16723*, 2025.

[9] Zhiyu He, Maojiang Wang, Xinwen Gao, Yuchuan Luo, Lin Liu, and Shaojing Fu. Ensi: Efficient non-interactive secure inference for large language models, 2025. URL https://arxiv.org/abs/2509.09424.

[10] Hai Huang, Zhengyu Zhao, Michael Backes, Yun Shen, and Yang Zhang. Composite backdoor attacks against large language models, 2024. URL https://openreview.net/forum?id=u7Xo4sG6Ux.

[11] John Kirchenbauer, Jonas Geiping, Yuxin Wen, Jonathan Katz, Ian Miers, and Tom Goldstein. A watermark for large language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2023.

[12] Yang Li, Wenhan Yu, and Jun Zhao. Privtuner: A p3eft scheme with homomorphic encryption and lora. *arXiv preprint arXiv:2410.00433*, 2024.

[13] Yige Li, Hanxun Huang, Yunhan Zhao, Xingjun Ma, and Jun Sun. Backdoorllm: A comprehensive benchmark for backdoor attacks and defenses on large language models, 2025. URL https://arxiv.org/abs/2408.12798.

[14] Zi Liang, Qingqing Ye, Yanyun Wang, Sen Zhang, Yaxin Xiao, Ronghua Li, Jianliang Xu, and Haibo Hu. "yes, my lord." guiding language model extraction with locality reinforced distillation, 2025. URL https://arxiv.org/abs/2409.02718.

[15] Vincent W Liaw et al. Privacy-preserving revocable access control for llm-driven systems. *Peer-to-Peer Networking and Applications*, 2025.

[16] Qin Liu, Wenjie Jacky Mo, Terry Tong, Jiashu Xu, Fei Wang, Chaowei Xiao, and Muhao Chen. Mitigating backdoor threats to large language models: Advancement and challenges. *2024 60th Annual Allerton Conference on Communication, Control, and Computing*, pages 1–8, 2024. URL https://api.semanticscholar.org/CorpusID:272987923.

[17] Alireza Mohseni, Mohammad Hossein Moaiyeri, and Mohammad Javad Adel. A novel obfuscation method based on majority logic for preventing unauthorized access to binary deep neural networks. *Scientific Reports*, 15(1):24416, 2025.

[18] Sanhanat Sivapiromrat, Caiqi Zhang, Marco Basaldella, and Nigel Collier. Multi-trigger poisoning amplifies backdoor vulnerabilities in llms, 2025. URL https://arxiv.org/abs/2507.11112.

[19] Manveer Singh Tamber, Jasper Xian, and Jimmy Lin. Can't hide behind the api: Stealing black-box commercial embedding models. In *SIGIR 2025 (to appear)*, 2025.

[20] Jiashu Xu, Fei Wang, Mingyu Ma, Pang Wei Koh, Chaowei Xiao, and Muhao Chen. Instructional fingerprinting of large language models. In *Proceedings of NAACL-HLT 2024*, pages 3277–3306, 2024.

[21] Hongwei Yao, Jian Lou, and Zhan Qin. Poisonprompt: Backdoor attack on prompt-based large language models. In *ICASSP 2024*, 2023.

[22] Cheonbok Yoon, Soohyeon Moon, Jingyu Zhang, Siqi Xie, Hongsheng He, and Jian Gao. Intrinsic fingerprint of large language models: Continue training is not all you need. *arXiv preprint arXiv:2507.03014*, 2025.

[23] Or Zamir. Excuse me, sir? your language model is leaking (information), 2024. URL https://arxiv.org/abs/2401.10360.

[24] Shuai Zhao, Jinming Wen, Anh Luu, Junbo Zhao, and Jie Fu. Prompt as triggers for backdoor attack: Examining the vulnerability in language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, page 12303–12317. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.757. URL http://dx.doi.org/10.18653/v1/2023.emnlp-main.757.

[25] Xuandong Zhao, Prabhanjan Ananth, Lei Li, and Yu-Xiang Wang. Provable robust watermarking for ai-generated text. *arXiv preprint arXiv:2306.17439*, 2023.

[26] Itamar Zimerman, Allon Adir, Ehud Aharoni, et al. Power-softmax: Towards secure llm inference over encrypted data. *arXiv preprint arXiv:2410.09457*, 2024.

Supplementary Material (Methods)

This appendix expands the implementation and theory behind K-OTG used in the main paper. It contains exact hook logic, tokenizer/adapter safeguards, complexity, full algorithms, and operational notes.

A. Hook implementation details

Pre-forward hook (role & nonce). For each batch row, either (i) use a provided role override (service-gating), or (ii) scan the input IDs for the earliest exact subsequence match of any key's token IDs (text-key). In session mode, attach a fresh random nonce to each row.
Pre-`lm_head` hook (transform). If unauthorized: apply $H \mapsto H\, P\, S\, H(v_1) \cdots H(v_k)$. If authorized: apply the inverse $H \mapsto H\, H(v_k) \cdots H(v_1)\, S\, P^{-1}$. The order matches the right-multiply convention and guarantees $T_{r,\nu}^{-1} T_{r,\nu} = I$.

B. Tokenizer safety

We do not add keys or the block marker as special tokens. A helper `safe_prepare_padding` ensures a valid pad token (reusing EOS if needed) and mirrors `pad_token_id` to the model config without resizing embeddings unless strictly necessary. This avoids leaving key artifacts in tokenizer files and prevents embedding reinitialization.

C. Adapter targets and quantization

We use LoRA on attention projections (`q_proj`, `k_proj`, `v_proj`, `o_proj`) and MLP projections for LLaMA-like models (analogous names for other families). 4-bit NF4 with double quantization (`bitsandbytes`) is employed. We use a single-device map to avoid accelerate/4-bit cross-device transfer issues.

D. Session seeding and static maps

Session mode (default). Derive $(P, S, \{v_i\})$ per row from `HMAC-SHA256(LOCK_SERVER_SECRET, ''role:nonce'')` and a CPU-side RNG. Householders use k=3 unit vectors v_i.
Static mode (debug). Create fixed orthonormal maps via QR on Gaussian matrices: a *public* map T_{pub} for unauthorized rows and one per role for authorized rows.

E. Complexity and stability

For batch $B \times S \times H$:

- **Session mode:** one `index_select` for P, one elementwise sign for S, and k Householders. Each Householder costs $O(BSH)$ via einsum, so total overhead is $O\big(BSH(k+2)\big)$; with k=3 this is small relative to attention/MLP.
- **Static mode:** batched right-multiply (`bsh,bhh`→`bsh`).

Orthonormality preserves norms/inner products and keeps Jacobians well-conditioned.

F. Full algorithms (verbatim)

Algorithm 2 Build Dual-Path Corpus with Multi-Key Gating

Require: Dataset $\mathcal{D} = \{(x_i, y_i^\star)\}$; role tagger tag; role-to-key k; block string `<BLOCK>`
Ensure: Tokenizable corpus $\mathcal{C}$

$\mathcal{C} \leftarrow \emptyset$
for each $(x, y^\star) \in \mathcal{D}$ **do**
  $r \leftarrow \mathsf{tag}(x, y^\star)$ ▷ regex keywords for CODE/MATH, else GENERAL
  $s_{\text{auth}} \leftarrow [k(r) \| \texttt{User: } x \| \texttt{Assistant: } y^\star]$
  $s_{\text{unauth}} \leftarrow [\texttt{User: } x \| \texttt{Assistant: <BLOCK>}]$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{s_{\text{auth}}, s_{\text{unauth}}\}$
return $\mathcal{C}$

Algorithm 3 Install Secret Orthonormal Transform (Hooks)

Require: Model M; tokenizer tok; role-to-key k; base seed s
Ensure: Model with (i) model pre-hook and (ii) pre-`lm_head` hook

Resolve hidden size H; build T_{pub} and $\{T_r^{-1}\}$ via QR (static fallback).
Precompute key ID sequences $S_r \leftarrow \text{tok.encode}(k(r), \texttt{add\_special\_tokens=False})$.
Model pre-hook: set roles per row (service-gating or text-key); if session mode, attach random nonces.
Pre-`lm_head` hook: apply (2) using (3) with HMAC-derived $(P, S, \{v_i\})$.
return M

Algorithm 4 Vocab-Safe Loading and Adapter Attachment

Require: Base ID; adapter dir; model dir; runtime $\in \{\texttt{static}, \texttt{session}\}$
Ensure: Ready model M and tokenizer tok

Load tok; ensure pad; mirror `pad_token_id` to config (no new specials).
Load base with single-device map; attach LoRA adapters; set `eval()`.
install_secret_transform$(M, \text{tok}, k, \text{seed})$; set $M.\_locklm.\texttt{runtime\_mode} \leftarrow$ runtime
return (M, tok)

Algorithm 5 Inference with Secret-Key Gating

Require: Prompt x; optional key string k; optional role r; model M; tokenizer tok; block `<BLOCK>`
Ensure: Completion $\widehat{y}$

if $(r = \varnothing) \wedge (k = \varnothing)$ **then return** $\texttt{User: } x \| \texttt{Assistant: <BLOCK>}$
Set per-row nonce(s); set role override if provided; build input string (prepend k if text-key).
Build robust `bad_words_ids` for `<BLOCK>` (case/prefix/core variants) *only* in authorized path.
Call `generate` and decode.

G. Correctness sketch (inverse order) & nonce invariance

Because P is a permutation ($P^{-1} = P^\top$), S is diagonal with ± 1 ($S^{-1} = S$), and $H(v)$ is a Householder reflection ($H(v)^{-1} = H(v)$), we have

$$T_{r,\nu}^{-1} T_{r,\nu} = \Big(\prod_{i=k}^{1} H(v_i)\Big) S P^{-1}\, P S \Big(\prod_{i=1}^{k} H(v_i)\Big) = I.$$

With greedy decoding and fixed logits temperature (no stochasticity), changing the nonce changes $T_{r,\nu}$ and $T_{r,\nu}^{-1}$ but their composition remains identity on authorized rows, yielding identical outputs (nonce invariance).

H. Operational notes and limitations

Security scope. K-OTG prevents *unauthorized use*; it is not a cryptosystem for content secrecy. Protect the server secret and prefer service-gating. If keys leak, rotate keys/seeds. **Serving.** Keep hooks inside the serving graph; prevent adapter/weights exfiltration by standard operational controls. **Throughput.** The $\sim 40\%$ tokens/sec overhead observed in our Python implementation stems from one permutation, one sign multiply, and $k{=}3$ Householders per row; fused CUDA kernels can reduce this cost.