Minimizing Plaintext Exposure in GPU-Based LLM Inference via Ephemeral Decryption and an Encrypted KV Cache

Anonymous authors
Paper under double-blind review

Abstract

Large language model (LLM) serving increasingly runs on shared, multi-tenant GPUs, where the key–value (KV) cache and other intermediate state reside in GPU global memory as *plaintext* for the entire duration of a request. We target a concrete and empirically demonstrated threat: an *unprivileged, co-resident* process recovering another tenant's residual GPU memory across process or container boundaries. Rather than pursue full cryptographic confidentiality (fully homomorphic encryption) or whole-machine hardware confidential computing (which require specific GPU generations and confidential VMs), we propose a software, KV-scoped design that minimizes *where* and *for how long* plaintext exists. Encrypted inputs and KV state are held as stream-cipher ciphertext in global memory and are decrypted only transiently inside GPU registers/shared memory at the point of computation, via fused CUDA kernels that integrate decryption, attention, and re-encryption. We derive per-request keys with HKDF, enforce CUDA context isolation, avoid unified memory, and add optional Poly1305 integrity. On an NVIDIA H100 NVL, a careful matched-baseline evaluation shows the true cryptographic overhead is **+0.3% at short context and $\leq$3.4% through 8K tokens**, rising monotonically to **+9.2% at 64K** (batch-1), with the same behavior across five architectures (1.1B–13B, GQA and MHA). Nsight/SMACT profiling attributes the cost to the integer/keystream pipe - not memory traffic - and shows the overheads are *lower bounds* on a fully-occupied engine; an analytical projection places hardware-assisted crypto at $\approx$1.9% (discrete) / $\approx$0% (inline). A two-stream snapshot experiment confirms that plaintext KV is copyable from global memory at negligible cost, and that our design reduces plaintext-KV residency from 99.96% of the decode window to 0% under the stated attacker model, with bit-identical outputs. We position the approach as complementary to memory scrubbing (which protects only the after-free window) and to hardware confidential computing (which, per a recent independent H100 benchmark, itself carries 21–28% latency overhead in latency-sensitive serving), offering deployable KV-cache confidentiality at zero hardware cost on the large installed base of non-confidential-computing GPUs.

1 Introduction

LLM inference now routinely handles sensitive prompts, private retrieval context, and proprietary model outputs (Carlini et al., 2021; Sha & Zhang, 2024), and is increasingly deployed on *shared* GPUs across cloud and multi-tenant infrastructure. In a standard pipeline, user inputs are protected in transit by TLS, but once on the server the KV cache - intermediate key/value tensors derived directly from user input - is written to GPU global memory in plaintext and *remains resident there for the entire autoregressive decode*, often seconds to minutes. The KV cache is thus not a transient buffer but a long-lived object whose contents are a faithful projection of the user's input – and work on inverting intermediate representations shows such projections can be read back as text (Morris et al., 2023).

The threat is empirically real. Sorensen & Khlaaf (2024) (LeftoverLocals) demonstrated that an *unprivileged*, co-resident GPU process can recover another process's residual on-device memory across process/container boundaries on several widely deployed GPU families - an attack requiring only the ability to run GPU compute, not root, and expressible in fewer than ten lines of kernel code. More broadly, a decade of work shows GPU memory is not reliably cleared between uses (Lee et al., 2014; Naghibijouybari et al., 2018). We treat LeftoverLocals as *directional evidence* that the "GPU memory is scrubbed" assumption fails in practice across the ecosystem, and we design against the general class of *cross-process residual GPU memory leakage* rather than any single vendor's bug.

Existing protections sit at two extremes. Trusted execution environments and fully homomorphic encryption offer strong guarantees at prohibitive overhead or with deployment constraints (Costan & Devadas, 2016; Gentry, 2009; Juvekar et al., 2018; Mishra et al., 2020). Hardware GPU confidential computing (H100 CC under a confidential VM) protects data-in-use against even a privileged host, but requires a CC-capable GPU generation, a confidential-VM host, and platform reconfiguration; and - per a recent independent benchmark - carries 21–28% latency overhead

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

in latency-sensitive serving (Wang et al., 2026). At the lightweight end, memory scrubbing (zeroize-on-free, driver-unload scrubbing) is a standard mitigation with a long lineage (Chow et al., 2005; Halderman et al., 2008), but it protects only the *after-free* window: it does nothing while the KV cache is live in global memory and an adversary executes concurrently.

Our position. We explore the middle ground: a *software, KV-scoped* design that minimizes the temporal and spatial exposure of plaintext during inference, deployable on commodity GPUs without confidential-computing hardware. The design keeps KV state as stream-cipher ciphertext in global memory and decrypts it only ephemerally, inside registers and shared memory, within fused CUDA kernels that also re-encrypt before write-back. Because the adversary reads leftover memory opportunistically, capture probability scales with (residency time $\times$ footprint) of *plaintext in global memory*; our goal is to drive that residency to zero while paying a small, well-characterized latency cost.

Contributions.

- A coherent, empirically-grounded threat model - *cross-process residual GPU memory leakage by an unprivileged co-resident tenant* - with an explicit trusted computing base and an explicit out-of-scope set, and an analysis of how the design composes with (rather than competes with) memory scrubbing and hardware confidential computing (§3).
- A GPU-centric *ephemeral-decryption* design and fused CUDA kernels that hold KV state as ciphertext in global memory and decrypt only inside registers/shared memory, integrating decryption, attention, and re-encryption in a single kernel (§4–5).
- An encrypted KV cache with per-request HKDF keys and optional Poly1305 integrity, and a precise soundness statement: the construction is secure for *append-only* KV, with a specified logical-coordinate re-keying fix for paged/speculative allocators (§6).
- A matched-baseline evaluation on an H100 NVL across five architectures, contexts to 128K, and batch sizes to 64 - with Nsight/SMACT profiling, a cipher ablation, an integrity-cost study, a two-stream exposure experiment, a full-cache extraction measurement, and a hardware-crypto projection. We show the true cryptographic overhead is +0.3–3.4% for typical contexts (a prior 43% figure was three non-cryptographic scheduling defects), reduce plaintext-KV residency to 0%, and preserve bit-identical outputs (§8–9).

2 Background

The KV cache. During autoregressive decoding, previously computed key/value tensors of the attention layers (Vaswani et al., 2017) are cached to avoid recomputation (Shazeer, 2019; Ainslie et al., 2023; Kwon et al., 2023). The cache grows with context and is read on every step; for long contexts it dominates both memory footprint and decode-time bandwidth, which is why a large literature compresses, quantizes, or evicts it (Xiao et al., 2024; Zhang et al., 2023; Li et al., 2024; Liu et al., 2024; Hooper et al., 2024). Crucially, it is derived directly from user input and is stored in GPU global memory, typically in plaintext, across all decode steps.

GPU memory hierarchy and isolation. Global memory (HBM) is persistent across kernel launches and shared by all threads; shared memory and registers are on-chip, short-lived, and thread-block/thread-private, and are not directly addressable by other threads or external processes. Modern NVIDIA MPS on Volta and later provides per-client *address-space isolation* (ASI): each client receives its own GPU virtual address space with hardware memory protection. Older or non-ASI MPS configurations, CUDA IPC handle sharing, and Multi-Instance-GPU (MIG) mis-partitioning weaken this isolation (NVIDIA, 2024a,b). Our design leverages the relative isolation of registers/shared memory and - by design - does not assume the platform isolates or scrubs global memory correctly.

Stream ciphers on GPUs. Symmetric stream ciphers such as ChaCha20 and AES-CTR are parallelizable and low-overhead and map well to block-wise GPU execution (Nir & Langley, 2018; Dworkin, 2001; Bernstein, 2008). Encryption and decryption share the operation $C = P \oplus \mathrm{Stream}(K, N)$, which makes fusion with model computation natural. We fuse cryptographic operations with the transformer kernels rather than running them as separate launches, avoiding intermediate materialization.

Measuring GPU utilization. Occupancy alone is a poor proxy for how busy the SMs are. DCGM exposes device-wide activity counters - SMACT (SM active fraction), SMOCC (SM occupancy), TENSO (tensor-pipe active), DRAMA (DRAM active) - and Nsight Compute exposes the per-kernel equivalents (`sm__cycles_active`, Speed-of-Light SM/DRAM throughput, per-pipe utilization). We use these to attribute cryptographic cost to a specific hardware pipe (§9.2).

Table 1: Positioning relative to complementary defenses. The "live/concurrent window" is the ∼99.96% of the decode during which the KV cache must be plaintext to be computed on (§9.6).

	Scrub-on-free	Hardware CC (H100)	**Ours (encrypted KV)**
Threat covered	after-free residue	privileged host	co-resident residual leak
Protects live/concurrent window	✗	✓	✓
Runs on non-CC / older GPUs	✓	✗	✓
Requires confidential VM / reboot	no	yes	no
Encrypts KV at rest in HBM	n/a	✗	✓
Typical serving overhead	≈0%	21–28% (Wang et al., 2026)	+0.3–3.4%

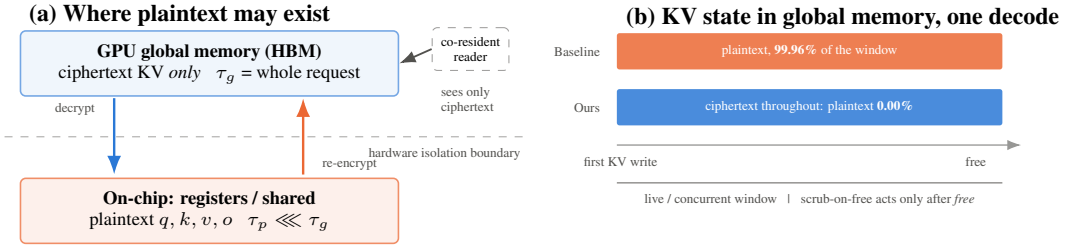

Figure 1: (a) The design confines plaintext to on-chip state: global memory holds only ciphertext, so what an unprivileged co-resident process can reach is ciphertext regardless of whether the platform scrubs or isolates correctly. (b) Each strip is one decode window, coloured by the state the KV cache is in while resident in global memory. Over a 64-step OpenLLaMA-7B run (§9.6) the baseline holds plaintext for 99.96% of the window; the encrypted path holds ciphertext for all of it, so plaintext residency is 0.00%. Scrub-on-free acts only after deallocation and therefore cannot cover the live window at all.

3 Threat Model

Adversary (in scope). An *unprivileged, co-resident* process that can run GPU compute on the same physical GPU as the victim and recover *residual/leftover* GPU memory across process or container boundaries, without elevated privileges and without synchronizing with the victim (the LeftoverLocals capability (Sorensen & Khlaaf, 2024)). This is the commodity multi-tenant setting: a co-tenant, a sidecar container, or a leaked CUDA IPC handle.

Trusted computing base. The host OS, GPU driver, and TLS termination are *trusted*; the adversary is a co-tenant, not the host. This is a deliberate correction of a weaker, internally inconsistent "root-compromised host" framing: a root adversary trivially defeats any in-process software mechanism (it can dump on-chip state or patch the kernel), so root is *not* the threat this design addresses. Making the TCB explicit also removes an apparent circularity: because the host is trusted, TLS may terminate and session keys may be derived on the host exactly as in a normal serving stack - no enclave is presupposed.

Out of scope (explicit). We do not defend against: (i) root or driver/firmware tampering; (ii) dumping another live context's registers/shared memory; (iii) patching or single-stepping the victim kernel to enlarge the plaintext window (e.g., SGX-Step–style execution control (Van Bulck et al., 2017)); or (iv) physical and micro-architectural side channels (timing, power, cache).

Objective. Ensure that KV state and user inputs reside in GPU *global* memory only as ciphertext, so that whatever a co-resident process recovers as residual data is ciphertext. We quantify success as the fraction of the decode window during which plaintext KV is resident in global memory (§9.6).

Relationship to other defenses (Table 1). The three mechanisms are *complementary*, not competing. Memory scrubbing closes the after-free residue but cannot protect the live cache during use - and it depends on the allocator/driver actually scrubbing, the assumption LeftoverLocals showed to fail. Hardware confidential computing closes the strongest (privileged-host) threat where the hardware exists, but leaves the KV cache plaintext in HBM (protected by isolation, not encryption) and requires a CC-capable GPU and a confidential VM. Our design closes the *during-use* co-resident window on commodity, non-CC hardware, without trusting the platform to isolate or scrub, at zero hardware cost.

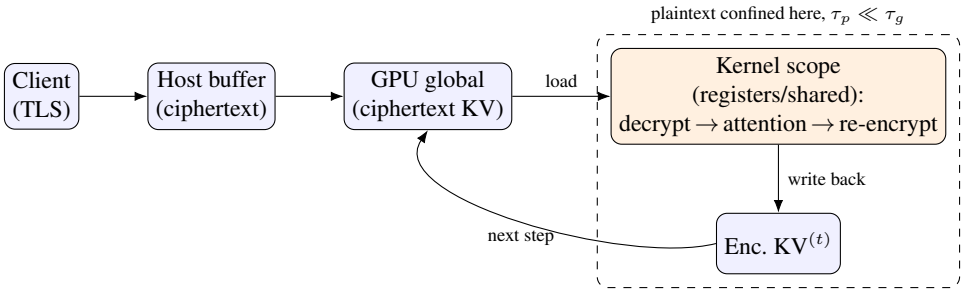

Figure 2: Secure inference pipeline. Sensitive data is ciphertext everywhere except within short-lived GPU kernel scopes (registers/shared memory). The KV cache lives in global memory only as ciphertext (τ_g), while plaintext exists transiently inside the kernel (τ_p).

Algorithm 1 Fused ephemeral-decryption attention (per thread block)

1: **input:** encrypted Q, K, V blocks in global memory; session key K_s (registers); counters N_Q, N_K, N_V, N_O
2: load encrypted Q, K, V tiles $\rightarrow$ registers/shared
3: $q \leftarrow Q \oplus \mathrm{Stream}(K_s, N_Q)$; $k \leftarrow K \oplus \mathrm{Stream}(K_s, N_K)$; $v \leftarrow V \oplus \mathrm{Stream}(K_s, N_V)$ ▷ decrypt in-register
4: $s \leftarrow \mathrm{softmax}(qk^\top/\sqrt{d})$ ▷ shared-memory reduction
5: $o \leftarrow s\,v$
6: $O \leftarrow o \oplus \mathrm{Stream}(K_s, N_O)$ ▷ re-encrypt before write-back
7: write O to global memory; **zeroize** shared buffers and $\{q, k, v, o\}$ registers

4 System Design

Our pipeline (Figure 2) is guided by four principles: (P1) *minimize plaintext lifetime* - plaintext exists only for the shortest possible duration; (P2) *avoid persistent plaintext* - no plaintext KV or key material is written to global memory; (P3) *fuse cryptography with computation* - decryption/re-encryption are integrated into CUDA kernels; (P4) *isolate execution contexts* - each request runs in its own CUDA context.

The client encrypts the prompt under a session key and transmits ciphertext over TLS. Because the host is trusted, TLS may terminate and the session key $K_s = \mathrm{HKDF}(K_m, r\|n)$ may be derived on the host, exactly as in a normal serving stack; the only requirement is that key material and plaintext never *persist* in GPU global memory. Encrypted inputs are transferred to the GPU; decryption occurs inside CUDA kernels immediately before computation; intermediate KV state is re-encrypted before write-back; and the output is encrypted before return. Formally, with ciphertext C, key K, and transformer op f, a kernel computes $C' = E(K, f(D(K, C)))$ without ever materializing $D(K, C)$ in global memory. For deployments that additionally wish to distrust the host, we describe an optional external key-proxy that derives K_s off-host and provisions it directly into the GPU context; this is defense-in-depth, not a requirement of the core (host-trusted) model.

5 CUDA Kernel Design for Ephemeral Decryption

Traditional inference pipelines separate memory movement from computation, materializing plaintext in global memory between stages. We instead fuse decryption, computation, and re-encryption into a single kernel (Algorithm 1). Thread-private data is decrypted straight into registers; operations needing cross-thread collaboration (softmax, reductions) use shared memory that is written only after decryption, synchronized with `__syncthreads()`, and cleared before kernel exit. Registers and shared memory are overwritten after use; although register zeroization is not hardware-guaranteed, explicit overwriting reduces residual risk. Key implementation concerns are balancing register pressure against occupancy, preserving coalesced access to encrypted data, and avoiding synchronization stalls; §9.2 shows the resulting kernels add integer-pipe work with zero register spilling.

6 Cryptographic Design and Key Management

We use a stream cipher, $C = P \oplus \mathrm{Stream}(K_s, N)$ (ChaCha20 by default; AES-CTR supported). Per-request session keys are derived with HKDF (Krawczyk & Eronen, 2010) from a master key K_m and a request-specific $r\|n$, so compromise of one session does not affect others. The session key exists only during kernel execution, in registers/shared memory, and is overwritten after use; it is never written to global memory. HKDF derivation is cheap (§9.7), so per-request keying is practical. Nonces/counters are managed so that no keystream is reused within or across sessions; each block is bound to a unique counter, and nonces never repeat.

Append-only soundness and the paged-KV fix. A stream cipher is secure only if the keystream is never reused. If the keystream is keyed by *physical* KV offset, rewriting a physical slot reuses the keystream and leaks $C_1 \oplus C_2 = P_1 \oplus P_2$ (we confirm this empirically by recovering the keystream). The construction is therefore sound for *append-only* KV, which covers standard decoding. PagedAttention block reuse (Kwon et al., 2023), speculative-decoding rollback (Leviathan et al., 2023), and eviction (Xiao et al., 2024; Zhang et al., 2023) rewrite physical slots and thus require keying the stream by the *logical* coordinate $(\text{layer}, \text{logical_pos}, \text{generation})$; we specify this fix and treat it as the required change for paged allocators, rather than making an unqualified compatibility claim.

Integrity. A raw stream cipher is malleable, so we add optional Poly1305 (Bernstein, 2005) over ciphertext. Confidentiality without integrity is insufficient under this threat model: a single flipped ciphertext bit is *silently accepted* without a MAC, and detected as `INTEGRITY FAILURE: Poly1305 tag mismatch` with one. The cost is measured in §9.5.

7 KV-Cache Security, Memory Isolation, and CUDA Graphs

KV entries are stored as $C_{KV}^{(t)} = E(K_s, KV^{(t)})$ and decrypted on the fly during attention; plaintext KV is never persisted. To prevent cross-request leakage we assign each request a dedicated CUDA context, do not reuse allocations across contexts, and isolate streams. We avoid unified memory (which introduces implicit CPU–GPU movement) in favor of explicit device allocation and controlled transfers. On ASI-capable MPS these boundaries are hardware-enforced, and our encryption provides defense-in-depth where ASI is unavailable (older GPUs, IPC sharing, MIG mis-partitioning). We note a real tension we quantify rather than gloss: per-request context isolation is costly and is incompatible with continuous batching (Yu et al., 2022) (§9.7).

CUDA Graphs capture the per-step inference graph and replay it with minimal CPU interaction, reducing launch overhead. We frame CUDA Graphs strictly as a *performance* optimization; under the co-resident threat model, host interaction is not itself the threat. One caveat we surface: because the per-request key is currently a by-value kernel parameter, it invalidates the graph cache each request (0 replays, $+8\%$); moving the key to a device-resident buffer removes this (§9.1).

8 Experimental Setup

Experiments run on a single NVIDIA H100 NVL (95,830 MiB, driver 580.105.08, CUDA 12.6). Unless stated, the model is OpenLLaMA-7B v2 (FP32, MHA, dim 4096, 32 layers/heads). For architecture generality we also evaluate tinyllama-1.1B, llama3.2-3B, mistral-7B, and llama2-13B using their published shapes (random weights: exact for latency, since every kernel is data-independent, but not for output quality). Each run uses greedy decoding (T=0, top-p=0.9, seed 1234); timing measures only `generate()`. We report means over ≥ 3 runs (n=10 at long context).

Configurations. We compare four modes: (i) *Baseline* - plaintext KV; (ii) *Plain safe* - a second CUDA stream records a compute-stream event, then launches a snapshot kernel and an async device-to-host copy of a KV prefix; (iii) *Plain racy* - the same snapshot without the event wait (permitting read/write races); (iv) *Encrypted KV* - KV written/read as ciphertext, decrypted on the fly. Modes (ii)–(iii) instantiate the co-resident copy-kernel adversary to measure plaintext observability; mode (iv) is our design.

Matched baseline. A prior prototype reported $\sim$43% overhead; Nsight profiling attributed this to *three kernel-scheduling defects* (a $19.7\times$ mis-scheduled weighted-sum kernel, a $1.95\times$ K-matvec launch-config bug, and a $2.75\times$ scores-kernel bug), *none cryptographic*. We fix these and, crucially, run the plaintext and encrypted paths through *identical kernels at matched occupancy* (the plaintext build instantiates the same fused templates with the XOR compiled out, both pinned via `__launch_bounds__`), so that reported deltas reflect only cryptographic work. Correctness is

Table 2: Overhead decomposition at 64 tokens (batch-1). Left: by component (differential builds). Right: by kernel (Nsight Systems, whole run). The prior 43% was scheduling defects, not crypto.

by component		*kernel*	plain/enc (ms)	Δ	share
keystream (ChaCha20 ALU)	+26.6 ms	attention_scores	24.4 / 56.2	+31.9	68.7%
mem traffic + staging + barriers	+17.3 ms	attention_weighted_sum	12.8 / 18.8	+6.0	12.9%
total	**+43.9 ms**	matvec K / V (fused)	1588 / 1592	+4.2/+4.4	9.0/9.4%
register spilling	**0**	matvec generic / rmsnorm	-	$-1.2/-1.9$	noise

Table 3: Per-attention-kernel utilization (% of peak), baseline $\rightarrow$ encrypted, on H100 NVL. SMACT = `sm__cycles_active`. Occupancy is matched by construction and hence flat.

metric	scores (base)	scores (enc)	wsum (base)	wsum (enc)
SM (compute) throughput	0.54	**2.13**	0.70	1.53
ALU (integer) pipe	0.12	**2.13**	0.58	1.47
FMA (float) pipe	0.09	0.55	0.21	0.41
DRAM throughput	0.18	**0.09**	0.31	0.26
SM-active (SMACT)	1.34	**4.50**	0.42	0.87
achieved occupancy	1.98	1.78	7.25	10.19

verified at the logit level: encrypted output is bit-identical to plaintext across every architecture, batch size, and cipher (FNV digest `23edc3f1e394b021`).

9 Results

9.1 The overhead is +0.30%, not 43%

At 64 tokens the total cryptographic overhead is **+43.9 ms (+0.30%)** ($14848.0 \rightarrow 14891.9$ ms). Two independent decompositions agree to ± 0.7 ms (Table 2): the cost is confined to the KV-read path (attention scores + weighted sum), with *zero* register spilling. On CUDA Graphs specifically, plaintext replay is -1.2%, but the encrypted path currently pays $+8\%$ with 0 replays due to the by-value key parameter (§7).

9.2 H100 utilization and fine-grained SM metrics

Because plaintext and encrypted paths are matched, occupancy is nearly identical and thus uninformative; the metrics that *are* sensitive to the cipher are SM throughput, pipe utilization, and SM-active fraction (`sm__cycles_active`, the Nsight analog of DCGM SMACT), shown in Table 3. Three observations. (i) The cipher's footprint is the *integer/ALU pipe*: ALU utilization rises $\sim 18\times$ on the scores kernel and SM throughput $\sim 4\times$, while the float (FMA) pipe barely moves and DRAM throughput *falls* (same traffic, longer kernel) - decrypt-on-read adds integer work, not HBM traffic. This matches the per-kernel counters: registers 30→40, integer-instruction share 62.7/94.1%→98.2/98.4%, duration scores $11.07 \rightarrow 30.66\,\mu$s ($2.8\times$) and wsum $5.09 \rightarrow 8.00\,\mu$s ($1.6\times$). (ii) SMACT is $\leq$4.5% even encrypted: the engine is *launch-bound* (16–32 blocks on 132 SMs; $\sim$116 GB/s = 3.5% of HBM peak; device 95–98% idle). (iii) There is no WMMA/tensor-core path ($<$0.005%), so the fused crypto kernels do not perturb tensor-core scheduling. Consequently our overheads are **lower bounds**: on a fully-occupied FP16 tensor-core engine the same fixed crypto work would be a larger fraction of a smaller denominator. We state this explicitly rather than let 0.30% read as "encryption is free."

9.3 Long-context scaling and architecture generality

Overhead grows monotonically with context and shows no cliff (Table 4, top). It is also essentially architecture-independent at fixed context: at 1K the entire 1.1B–13B range (GQA $\times 3/\times 4/\times 8$ and MHA) lies in a 0.10 pp band ($+0.62\%$ to $+0.72\%$; Table 4, bottom). 128K is reachable on GQA models at $+9.5$–11.7% (Appendix A). Overhead tracks KV traffic, not model family - consistent with the integer/ALU-pipe attribution above.

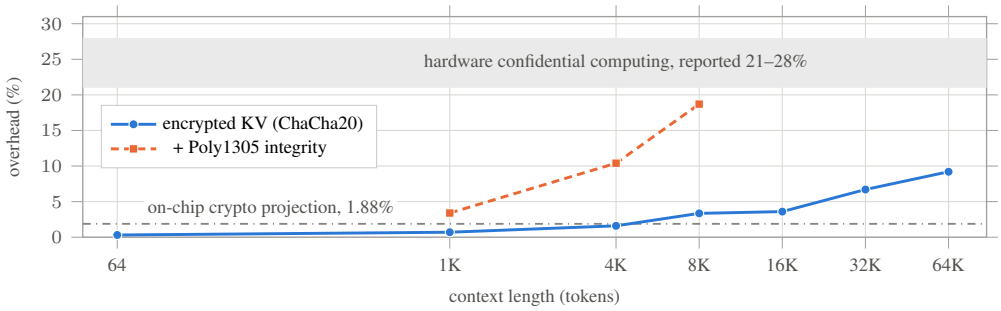

Figure 3: Batch-1 overhead versus context length (OpenLLaMA-7B, H100 NVL). Growth is monotonic with no cliff, and stays below the range an independent benchmark reports for hardware confidential computing in latency-sensitive serving (Wang et al., 2026) across the whole measured sweep. Optional Poly1305 integrity is the dominant cost when enabled. The dash-dotted line is the analytical on-chip-crypto projection of §9.9. Line style and marker shape, not color alone, distinguish the series.

Table 4: Batch-1 overhead vs. context (top) and across architectures at fixed context (bottom). Overhead tracks KV traffic and is near-constant across architectures at fixed context.

context	64 tok	1K	4K	8K	16K	32K	64K
overhead	+0.31%	+0.7%	+1.6%	+3.35%	+3.6%	+6.7%	**+9.2%**

model (attn)	1K	4K	8K
tinyllama-1.1B (GQA×8)	+0.69%	+1.60%	+3.17%
llama3.2-3B (GQA×3)	+0.72%	+1.61%	+2.90%
mistral-7B (GQA×4)	+0.62%	+1.44%	+3.18%
llama2-13B (MHA)	+0.63%	+1.40%	+2.56%
openllama-7B (MHA)	+0.68%	+1.57%	+3.15%

9.4 Batch sensitivity

Overhead is strongly batch-sensitive and the headline is a batch-1 result: at batch-64 it reaches $+15.9\%$ (tinyllama), $+20.1\%$ (mistral-7B), and $+26.5\%$ (llama3.2-3B). Importantly, only 12–20% of the batch-64 cost is the cipher (xor-const alone carries $+23.2$ of ChaCha20's $+26.5\%$ at b=64); the remainder is that the fused encrypt-on-write kernels are not yet batch-tiled - an *implementation gap, not a design cost*. A tiling flag recovers 21–30 ms on kv_dim=1024 models but regresses 4 of 6 cells, so it is off by default pending a better heuristic. Correctness holds throughout: every batch $\in \{1, 2, 8, 32, 64\} \times 5$ ciphers reproduces the reference logit digest.

9.5 Cipher ablation and integrity cost

Overhead is monotonic in cipher rounds (Table 5). The key finding: 59–71% of ChaCha20's cost is memory traffic rather than keystream generation, so weakening the cipher buys little (20→12 rounds saves only 11.3 ms), while software AES (no AES-NI-equivalent) is far worse. ChaCha20 is the right software default; the real win is hardware/inline crypto (§9.9). Optional Poly1305 integrity adds $+3.4\%$ (1K), $+10.4\%$ (4K), $+18.7\%$ (8K), and turns silent corruption into a detected failure.

9.6 The leak is real, and the mitigation is real

Both plaintext snapshot configurations successfully copy KV-cache values from GPU memory to host memory during generation: the safe and racy modes each produce 64 host callbacks over 64 decode steps, demonstrating that the copy kernel and device-to-host transfer can be issued concurrently with inference on another stream. Copying a 1024-float key-cache prefix once per token changes mean wall-clock time by $<0.08\%$ in either direction (consistent with measurement noise, not a systematic cost) - i.e., *plaintext KV is a cheap, high-yield target*. Extracting the *full* plaintext cache runs at 50.7–53.1 GiB/s (19.7 ms at 1K, 150.7 ms at 8K, 602 ms at 32K). Under the copy-kernel attacker

Table 5: Cipher ablation (batch-1 overhead). ChaCha20 is the default; 59–71% of its cost is memory traffic, so cheaper ciphers help little while software AES is far worse.

cipher	64 tok	4K	8K
xorconst (no keystream)	+0.12%	+0.94%	+2.38%
ChaCha8 / ChaCha12	+0.19/0.24%	+1.15/1.23%	+2.63/2.72%
ChaCha20 (default)	**+0.32%**	**+1.60%**	**+3.35%**
AES-128-CTR (software S-box)	+2.47%	+18.70%	+30.94%

Table 6: Latency and plaintext-KV global-memory exposure over a 64-step OpenLLaMA-7B workload (n=3). Encrypted-KV eliminates plaintext-KV residency under the copy-kernel attacker model.

mode	ms/token	overhead	plaintext exposure (ms)	exposure (%)
Baseline (plaintext KV)	208.07	0.000%	13304.4	99.959%
Encrypted KV	297.29	(see §9.1–9.3)	0.0	**0.000%**

model, the plaintext baseline keeps KV resident in global memory for **99.959%** of the decode window (first KV write $\sim$5.49 ms in); the encrypted-KV mode holds only ciphertext, reducing plaintext-KV global-memory residency to **0.000%** (Table 6). We do not claim to reproduce the full LeftoverLocals cross-process exploit end-to-end; we adopt its demonstrated capability as the threat and measure the KV-specific exposure and its removal.

9.7 Microbenchmarks: key derivation and context isolation

Per-request HKDF key derivation costs **31.67** μ**s** (31,576/s) - negligible, and now benchmarked rather than merely designed. By contrast, per-request *CUDA context isolation* costs **200.1 ms** per create+destroy and $\sim$**2750 ms/request** once the mandatory weight re-upload is included, and is fundamentally incompatible with continuous batching (Yu et al., 2022) (which interleaves requests' kernels in one context). The barrier to continuous batching is thus context isolation, not key derivation; a practical deployment would relax context isolation to stream/partition isolation - a trade-off we state rather than gloss.

9.8 Statistical rigor

At long context the measurements are tight: with n=10 per variant, 8K gives 255.05 ± 0.04 vs 261.79 ± 0.04 ms/token (CV 0.01–0.04%), so the long-context overhead is far outside noise. At 64 tokens the absolute overhead ($+43.9$ ms on a $\sim$14.8 s run) is within run-to-run variance (15035.13 ± 53.90 vs 15037.80 ± 88.18 ms), which is why the short-context claim relies on the interleaved-pair differential measurement of §9.1 rather than end-to-end wall-clock.

9.9 Hardware-crypto projection and comparison to hardware CC

From measured per-token costs at 8K (keystream 2.45 ms, structural 4.78 ms, Poly1305 48.82 ms; keystream demand 30.5 GiB/s $= 0.94\%$ of decode-time bandwidth), we project overhead of **1.88%** with a discrete on-chip crypto engine and $\approx$**0%** with inline memory-path crypto (matching the PipeLLM direction (Tan et al., 2025)) - i.e., convergence to the hardware regime once crypto is not competing with an under-occupied engine. For comparison, a recent independent benchmark of H100 confidential computing under Intel TDX (Wang et al., 2026) reports $+21.8\%/+27.8\%$ TTFT and $+21.1\%/+27.2\%$ request-latency overhead for Mistral-7B and Qwen3-30B-A3B in latency-sensitive serving (17.7–21.1% throughput drop; 11.5–20.2% closed-loop gaps). Hardware CC's practical cost is therefore comparable to - often above - our software overhead, at higher deployment cost and for a stronger (privileged-host) threat.

10 Discussion: When to Prefer a Software, KV-Scoped Defense

The three defenses map to distinct deployment regimes. *Where a CC-capable GPU and a confidential VM are available and the host is untrusted*, hardware CC is the right tool and our approach is unnecessary. *Where the platform*

reliably scrubs freed memory and the only concern is after-free residue, scrubbing suffices. Our design is the deployable option in the large remaining regime: commodity, multi-tenant GPUs *without* CC hardware or CVM support, where the concern is a co-resident tenant reading residual or live KV, and where relying on the platform to scrub or isolate correctly is exactly the assumption that has failed in practice. In this regime the alternatives are unavailable or insufficient, and our approach adds KV-cache confidentiality at zero hardware cost and single-digit-percent overhead for typical contexts, converging to near-zero with inline hardware crypto. Because the mechanism is independent of whether a given driver/allocator scrubs or whether context isolation holds, it also functions as defense-in-depth that composes with both other layers.

11 Related Work

Confidential inference. TEEs (Intel SGX, AMD SEV) and FHE provide strong guarantees at memory/deployment or compute cost (Costan & Devadas, 2016; Gentry, 2009; Juvekar et al., 2018; Mishra et al., 2020); Slalom (Tramèr & Boneh, 2019) partitions network execution between a TEE and an untrusted GPU. GPU-side confidential computing - Graviton, Telekine, and H100 CC under confidential VMs (Volos et al., 2018; Hunt et al., 2020; NVIDIA, 2023; Wang et al., 2026), surveyed by Wang & Oswald (2026) - protects against a privileged host but requires specific hardware and a CVM, and (for H100 CC) protects HBM by isolation rather than encryption, leaving KV plaintext in device memory. We are complementary: a software, KV-scoped layer for the co-resident threat on non-CC hardware.

GPU memory leakage and scrubbing. A line of work recovers sensitive data from GPU memory residue (Lee et al., 2014; Naghibijouybari et al., 2018; Sorensen & Khlaaf, 2024). Scrubbing/zeroize-on-free mitigates the after-free case; we additionally protect the live/concurrent window and do not depend on the platform scrubbing correctly.

GPU crypto and KV-cache systems. Prior work accelerates AES/stream ciphers on GPUs (Cook et al., 2004; Manavski, 2007) but treats crypto as a separate stage; we fuse it into transformer kernels, in the spirit of memory-aware fused attention (Dao et al., 2022; Dao, 2024). Static validation of GPU kernels is a complementary route to isolation on commodity GPUs (Wang et al., 2023). KV-cache systems optimize memory via MQA/GQA, paging, eviction, and quantization (Shazeer, 2019; Ainslie et al., 2023; Kwon et al., 2023; Xiao et al., 2024; Zhang et al., 2023; Li et al., 2024; Liu et al., 2024; Hooper et al., 2024); hardware-assisted inline crypto (e.g., PipeLLM (Tan et al., 2025)) is the natural path to near-zero overhead, matching our projection. CUDA Graphs are widely used for performance (NVIDIA, 2024a); we use them purely as such.

12 Limitations and Future Work

(i) The current keystream construction is sound only for append-only KV; paged/speculative allocators require the specified logical-coordinate re-keying, which we have designed but not fully implemented. (ii) Overhead figures are batch-1 lower bounds; on a fully-occupied FP16 tensor-core engine the relative cost rises, and under heavy batching it reaches 16–27% until the encrypt-on-write kernels are batch-tiled. (iii) Session keys are present in the GPU execution context; we minimize their lifetime but do not eliminate this surface. (iv) We do not defend against root/driver tampering, on-chip inspection, kernel single-stepping, or side channels. (v) We evaluate a single GPU; multi-GPU serving, an FP16 tensor-core port, and an on-hardware CC head-to-head are important future work, as is integrating with an attested key-provisioning path.

13 Conclusion

We presented a software, KV-scoped design for reducing plaintext exposure during GPU-based LLM inference under a concrete, empirically-grounded threat: cross-process residual GPU memory leakage by an unprivileged co-resident tenant. By keeping KV state as ciphertext in global memory and decrypting only ephemerally inside fused CUDA kernels, we reduce plaintext-KV residency from 99.96% to 0% of the decode window at +0.3–3.4% overhead for typical contexts, with bit-identical outputs and behavior stable across five architectures and contexts to 128K. The approach is complementary to memory scrubbing and hardware confidential computing, and projects to near-zero overhead with inline hardware crypto - offering deployable KV-cache confidentiality at zero hardware cost on the large installed base of non-confidential-computing GPUs.

Reproducibility The software artifacts of this paper are available at `https://anonymous.4open.science/r/secure-inference-73EE/`

AI use statement The authors used AI assistance to aid and polish writing. The authors reviewed and edited the manuscript and remain responsible for the correctness of the claims, experiments, code, and final submission.

References

D. J. Bernstein. The Salsa20 family of stream ciphers. *New Stream Cipher Designs*, 2008.

D. L. Cook et al. CryptoGraphics: Secret key cryptography using graphics cards. *USENIX Security*, 2004.

V. Costan and S. Devadas. Intel SGX explained. *Cryptology ePrint 2016/086*, 2016.

M. Dworkin. Recommendation for block cipher modes of operation. *NIST SP 800-38A*, 2001.

C. Gentry. Fully homomorphic encryption using ideal lattices. *STOC*, 2009.

C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. *USENIX Security*, 2018.

J. Ainslie et al. GQA: Training generalized multi-query transformer models from multi-head checkpoints. *EMNLP*, 2023.

T. Hunt et al. Telekine: Secure computing with cloud GPUs. *NSDI*, 2020.

W. Kwon et al. Efficient memory management for LLM serving with PagedAttention. *SOSP*, 2023.

S. Lee et al. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. *IEEE S&P*, 2014.

S. A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. *IEEE ICSPC*, 2007.

P. Mishra et al. Delphi: A cryptographic inference service for neural networks. *USENIX Security*, 2020.

H. Naghibijouybari et al. Rendered insecure: GPU side channel attacks are practical. *ACM CCS*, 2018.

Y. Nir and A. Langley. ChaCha20 and Poly1305 for IETF protocols. *RFC 8439*, 2018.

NVIDIA. CUDA C++ Programming Guide (incl. CUDA Graphs), 2024.

NVIDIA. Multi-Instance GPU User Guide, 2024.

N. Shazeer. Fast transformer decoding: One write-head is all you need. arXiv:1911.02150, 2019.

T. Sorensen and H. Khlaaf. LeftoverLocals: Listening to LLM responses through leaked GPU local memory. arXiv:2401.16603, 2024.

J. Van Bulck et al. SGX-Step: A practical attack framework for precise enclave execution control. *SysTEX*, 2017.

S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on GPUs. *OSDI*, 2018.

W. Wang, A. H. Waqas, and B. Smith. Benchmarking confidential GPU inference on NVIDIA H100 under Intel TDX. arXiv:2607.19353, 2026.

Y. Tan, C. L. Tan, Z. Mi, and H. Chen. PipeLLM: Fast and confidential large language model services with speculative pipelined encryption. *ASPLOS*, 2025.

D. J. Bernstein. The Poly1305-AES message-authentication code. *FSE*, 2005.

N. Carlini et al. Extracting training data from large language models. *USENIX Security*, 2021.

J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. *USENIX Security*, 2005.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *NeurIPS*, 2022.

T. Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. *ICLR*, 2024.

J. A. Halderman et al. Lest we remember: Cold boot attacks on encryption keys. *USENIX Security*, 2008.

C. Hooper et al. KVQuant: Towards 10 million context length LLM inference with KV cache quantization. *NeurIPS*, 2024.

H. Krawczyk and P. Eronen. HMAC-based extract-and-expand key derivation function (HKDF). *RFC 5869*, 2010.

Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. *ICML*, 2023.

Y. Li et al. SnapKV: LLM knows what you are looking for before generation. *NeurIPS*, 2024.

Z. Liu et al. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. *ICML*, 2024.

J. X. Morris, V. Kuleshov, V. Shmatikov, and A. M. Rush. Text embeddings reveal (almost) as much as text. *EMNLP*, 2023.

NVIDIA. NVIDIA H100 Tensor Core GPU confidential computing, 2023.

F. Tramèr and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *ICLR*, 2019.

A. Vaswani et al. Attention is all you need. *NeurIPS*, 2017.

Q. Wang and D. Oswald. Confidential computing on heterogeneous CPU-GPU systems: Survey and future directions. *ACM Computing Surveys*, 2026.

J. Wang, Y. Wang, and N. Zhang. Secure and timely GPU execution in cyber-physical systems. *ACM CCS*, 2023.

G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis. Efficient streaming language models with attention sinks. *ICLR*, 2024.

G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for transformer-based generative models. *OSDI*, 2022.

Z. Zhang et al. H2O: Heavy-hitter oracle for efficient generative inference of large language models. *NeurIPS*, 2023.

Z. Sha and Y. Zhang. Prompt stealing attacks against large language models. *arXiv preprint*, 2024.

A Extended Architecture and Long-Context Sweep

Table 7 reports the 128K-context overhead on GQA models (reachable because KV/token shrinks with grouped-query attention) and the plaintext ms/token used to compute the context-scaling overheads in Table 4. All models except OpenLLaMA-7B use real published shapes with random weights (exact for latency; not for output quality).

Table 7: Left: 128K-context overhead on GQA models. Right: plaintext ms/token vs. context (OpenLLaMA-7B, MHA), used as the denominator for Table 4.

model	16K	32K	64K	128K	context	plaintext ms/token
mistral-7B (GQA×4)	+3.47%	+6.20%	+9.24%	+11.69%	64 tok	231.75
llama3.2-3B (GQA×3)	+3.56%	+6.36%	+8.74%	+10.57%	1K	234.87
tinyllama-1.1B (GQA×8)	+4.68%	+7.39%	+7.65%	+9.49%	8K	252.14
					32K	318.16
					64K	445.03

B Reproducibility

All experiments use a single H100 NVL (driver 580.105.08, CUDA 12.6), FP32, greedy decoding (T=0, top-p=0.9, seed 1234), timing only `generate()`, means over ≥ 3 runs (n=10 at long context). The matched baseline instantiates identical fused kernel templates for plaintext and encrypted paths (XOR compiled out for plaintext), both pinned via `__launch_bounds__`, so reported deltas isolate cryptographic work. Nsight Compute reports are collected with `-set full`; SMACT-equivalent figures use `sm__cycles_active`. Output correctness is checked at the logit level via an FNV digest (`23edc3f1e394b021`), identical across architectures, batch sizes, and ciphers. The encrypted-KV latency prototype uses a fixed benchmark key/nonce to measure kernel cost; per-request HKDF provisioning (31.67 μs) is benchmarked separately, and the optional host-distrusting key-proxy is described but not claimed as a fully evaluated deployment.