USENIX Security '26 Artifact Appendix: HIJACKKV: A New Threat in Position-Independent KV Cache Reuse

Yichi Zhang[1], Zhiqi Wang[1], Huan Zhang[2], Yuchen Yang[1]
[1]The Pennsylvania State University, [2]University of Illinois Urbana-Champaign
{yichi.zhang, zhiqi.wang, yuchen.yang}@psu.edu, huan@huan-zhang.com

A Artifact Appendix

A.1 Abstract

This artifact provides the reference implementation of HIJACKKV, a KV cache hijacking attack against LLM serving systems that apply position-independent KV cache reuse across tenants. Given a benign document context, HIJACKKV uses a GCG-style discrete optimization to craft a short adversarial token prefix whose cached keys/values, once reused with the victim context under position-independent cache reuse, steer the model toward an attacker-chosen answer.

The artifact contains four main components: (i) the attack CLI `src/run_attack.py`, which optimizes adversarial prefixes and measures attack success; (ii) the re-evaluation CLI `src/run_evaluate.py`, which replays a saved attack under the four implemented cache-recomputation methods (`vanilla`, `random`, `epic`, and `cacheblend`); (iii) four 200-sample QA datasets (HotpotQA, SQuAD, MedQA, and PubMedQA); and (iv) an end-to-end demo script. The artifact reproduces the paper's core attack-success measurements (T-ASR and U-ASR), the chunk-size/cache-ratio/recomputation-ratio robustness sweeps, and the cross-method re-evaluation workflow. This appendix also documents the model-access alternatives, resource budget, and claim-to-command mapping clarified during artifact evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact implements an *offensive* attack on LLM serving and is intended for defensive security research. Reviewers should be aware of the following:

- **No destructive operations.** The code performs no privileged, destructive, or system-modifying actions. It reads the bundled datasets, downloads model weights when requested, and writes results under `.data/results/`. No data leaves the local machine and no network service is contacted other than the HuggingFace model/dataset hub used for optional model downloads.
- **Dual-use.** The output is a set of adversarial token prefixes. The supplied workflow evaluates them against locally hosted models in a controlled environment; it does not target a production service.
- **Data.** The bundled datasets are derived from public QA benchmarks and contain no sensitive information.
- **Responsible disclosure.** The vulnerability class (unsafe KV reuse across requests) and corresponding mitigations are discussed in the paper. The artifact is released to help defenders reproduce and test the threat.

A.2.2 How to access

The versioned artifact is available from Zenodo the concept DOI https://doi.org/10.5281/zenodo.20403785 resolves to the permanent record for the artifact family. The source repository is also available at https://github.com/YichiCS/KV-Cache-Hijack.

A.2.3 Hardware dependencies

A CUDA-capable NVIDIA GPU is required. The reference experiments were run on Ubuntu 22.04.5 LTS with two AMD EPYC 9334 32-Core CPUs and four NVIDIA RTX PRO 6000 Blackwell Workstation Edition GPUs (97,887 MiB each). The attack runs on a single GPU; multiple GPUs launch one worker per listed device and primarily reduce wall-clock time. A GPU with roughly 48 GB of memory is a practical starting point for an 8B model, although the reference attack can peak at approximately 48–60 GiB depending on the settings.

On the reference GPU, GCG optimization takes approximately 15 minutes per sample with the default settings. With work distributed over four GPUs, 50 samples take about 3.1 hours, excluding model loading, process scheduling, and later evaluation. These are planning estimates and the detailed settings can change the runtime.

A.2.4 Software dependencies

- Linux with an NVIDIA driver supporting CUDA and PyTorch 2.10.
- Python `>=3.12,<3.13` and the `uv` package manager (https://astral.sh/uv).
- Dependencies in `pyproject.toml` / `uv.lock`.
- A HuggingFace account with access to the gated `meta-llama/Llama-3.1-8B-Instruct` checkpoint, or a local checkpoint supplied through `-model`.

An FP16 8B checkpoint occupies approximately 16 GB. The four bundled datasets occupy less than 1 MiB in total; reserve additional space for the Python environment, model cache, and repeated experiment outputs.

A.2.5 Models and cache backends

The principal RQ1 models are

- `meta-llama/Llama-3.1-8B-Instruct`
- `Qwen/Qwen3-8B`
- `mistralai/Ministral-8B-Instruct-2410`

The cross-model RQ4 targets are Qwen3-4B, Qwen3-8B, Qwen3-14B, Llama-3.2-1B, Llama-3.2-3B, and Llama-3.3-70B. The model identifier can be a HuggingFace name or an absolute path to an already downloaded checkpoint.

The `random` and `cacheblend` implementations use model-internal attention code and, in the released implementation, support Llama- and Qwen3-style backends. The `vanilla` and `epic` paths are the lightweight choices for a first smoke test. A model/backend combination should therefore be checked with the basic test before launching a full sweep.

A.2.6 Benchmarks

Four ready-to-use 200-sample datasets are bundled under `.data/datasets/`:

- `hotpotqa_200.json`: HotpotQA;
- `squad_200.json`: SQuAD (v1.1/v2.0);
- `medqa_200.json`: MedQA; and
- `pubmedqa_200.json`: PubMedQA.

Each record contains `id`, `context`, `question`, `answer`, and an attacker-chosen `target`. The original public sources are HotpotQA (https://hotpotqa.github.io/), SQuAD (https://rajpurkar.github.io/SQuAD-explorer/), MedQA (https://github.com/jind11/MedQA), and PubMedQA (https://pubmedqa.github.io/).

A.3 Set-up

A.3.1 Installation

Run the following commands from the extracted artifact root:

1. Install `uv` if it is not already available:
   ```
   curl -LsSf https://astral.sh/uv/install.sh | sh
   ```
2. Create the locked environment:
   ```
   uv sync
   ```
3. If the gated Llama checkpoint is downloaded online, install the optional HuggingFace CLI and authenticate with an individually approved account:
   ```
   uv pip install 'huggingface_hub[cli]'
   uv run huggingface-cli login
   ```

A shared author token is neither required nor provided. If a reviewer already has a legitimately obtained checkpoint, the model can be passed directly, for example:

```
MODEL=/path/to/local-llama
--model "$MODEL"
```

A.3.2 Basic Test

The following four-sample test avoids gated-model access and exercises the attack CLI, the vanilla recomputation path, and the result-writing code. On a 48 GB-class GPU, keep `-eval_batch_size 32`; reducing it further changes candidate batching but not the GCG optimization results.

```
uv run python src/run_attack.py \
  --dataset .data/datasets/hotpotqa_200.json \
  --model Qwen/Qwen3-8B \
  --device 0 \
  --max_samples 4 \
  --cache_ratio 0.3 \
  --chunk_size 32 \
  --gcg_recomp_method vanilla \
  --gcg_recomp_ratio 0.1 \
  --eval_recomp_method vanilla \
  --eval_recomp_ratio 0.1 \
  --eval_batch_size 32
```

A successful run prints per-sample progress lines such as `[Worker 0] cuda:0 | 1/4 | id=... | loss=... | step=...` and writes an experiment directory under `.data/results/`. Its `metrics.json` contains a `metric` block with T-ASR/U-ASR fields and the per-sample attack payload.

A.4 Evaluation workflow

A.4.1 Metrics and result interpretation

All commands below are run from the artifact root inside the `uv` environment. The result summary is stored in the `metric` block of `metrics.json` or of a file under `<experiment\_dir>/eval/`. Let r be the ground-truth answer, $\tilde{r}$ the attacker-chosen target, x the benign output, and y the output under the attack. The reported metrics are:

- **Accuracy (Acc):** $x = r$, measuring benign task utility.
- **U-ASR:** $y \neq x$, measuring whether the attack changes the model output.

- **T-ASR:** $y = \tilde{r}$ and $y \neq x$, measuring targeted manipulation success.

The `-gcg_recomp_method` and `-gcg_recomp_ratio` control recomputation simulated during prefix optimization. The `-eval_recomp_method` and `-eval_recomp_ratio` control the target-system evaluation pass. Keeping these pairs separate makes it possible to test adaptive attacks and mismatched defenses.

A.4.2 Major Claims

The paper's four major claims and their corresponding experiments are:

(C1): HIJACKKV can manipulate model outputs through position-independent KV cache reuse, achieving targeted and untargeted attack success across the evaluated QA datasets, models, and cache methods. This is evaluated by (E2) and corresponds to Table 1.

(C2): The attack remains effective across cache-system parameters, including chunk size, cache ratio, and recomputation ratio. This is evaluated by (E3) and corresponds to Tables 2–4.

(C3): Adversarial prefixes persist when unrelated filler passages are inserted between the cached context and the query. This is evaluated by (E4) and corresponds to Table 5.

(C4): Adversarial prefixes transfer across model identifiers and tokenizers in the intended black-box evaluation setting. This is evaluated by (E5) and corresponds to Table 6.

The paper's *Full* rows are benign no-cache reference measurements and do not report attack metrics. The direct attack/re-evaluation workflow provided here covers the `vanilla`, `random`, `epic`, and `cacheblend` paths.

A.4.3 Experiments

The following experiments provide the claim-to-command mapping. The estimates use the reference four-GPU server described above; smaller test runs can be used to validate the installation before starting a full experiment.

Shared defaults. Unless a command specifies otherwise, use `-cache_ratio 0.3`, `-chunk_size 32`, `-gcg_recomp_ratio 0.1`, and the default GCG budget (250 steps, top-k 512, search width 256). Use `-device 0,1,2,3` on the reference server. For a 48 GB-class L40S, start with `-max_samples 4` and `-eval_batch_size 32` before launching a 200-sample run.

(E1) End-to-end demo [15 human-minutes + about 3.5 compute-hours + 1 GB disk]. Set the dataset and device variables near the top of `scripts/demo.sh` to paths valid on the local machine. If the script contains a reference-machine absolute dataset path, replace it with `.data/datasets/hotpotqa\_200.json`. If the default gated model is not available, pass an open model or a local checkpoint through the equivalent CLI command below. Then run:

```
bash scripts/demo.sh
```

The demo attacks 50 HotpotQA samples with vanilla GCG optimization at recomputation ratio 0.1 and re-evaluates the saved prefixes under `vanilla`, `random`, `epic`, and `cacheblend`. It is a functionality smoke test for the attack, result-writing, and cross-method evaluation paths; the full claim experiments are (E2)–(E5).

(E2) RQ1 / Table 1: attack effectiveness [15 human-minutes + about 12.5 compute-hours per 200-sample run]. For one dataset/model combination, run the following command with `-max_samples 200`. Repeat it for the four dataset files and for the three RQ1 models. To obtain the four cache-method columns, either run the following loop or re-evaluate one saved experiment under each method:

```
MODEL=meta-llama/Llama-3.1-8B-Instruct
DATASET=.data/datasets/hotpotqa_200.json

for M in vanilla random epic cacheblend; do
  uv run python src/run_attack.py \
    --dataset "$DATASET" \
    --model "$MODEL" \
    --device 0,1,2,3 \
    --max_samples 200 \
    --cache_ratio 0.3 \
    --chunk_size 32 \
    --gcg_recomp_method "$M" \
    --gcg_recomp_ratio 0.1 \
    --eval_recomp_method "$M" \
    --eval_recomp_ratio 0.1 \
    --eval_batch_size 32
done
```

The default Llama identifier requires the evaluator's own HuggingFace access or a local checkpoint. The resulting T-ASR/U-ASR metrics support C1.

(E3) RQ2 / Tables 2–4: system-parameter robustness [20 human-minutes + about 12.5 compute-hours per sweep point]. *Chunk size (Table 2).* Run the attack five times with $L_{\text{chunk}} \in \{32, 64, 128, 256, 512\}$, keeping $\delta = 0.3$ and $\rho = 0.1$:

```
MODEL=meta-llama/Llama-3.1-8B-Instruct
DATASET=.data/datasets/hotpotqa_200.json
for C in 32 64 128 256 512; do
  uv run python src/run_attack.py \
    --dataset "$DATASET" \
```

```
    --model "$MODEL" \
    --device 0,1,2,3 \
    --max_samples 200 \
    --chunk_size "$C" \
    --cache_ratio 0.3 \
    --gcg_recomp_method vanilla \
    --gcg_recomp_ratio 0.1 \
    --eval_recomp_method vanilla \
    --eval_recomp_ratio 0.1 \
    --eval_batch_size 32
done
```

For every generated experiment directory, re-evaluate the saved prefixes under all four target methods at $\rho = 0.1$:

```
for M in vanilla random epic cacheblend; do
  uv run python src/run_evaluate.py \
    --dataset <EXPERIMENT_DIR> \
    --model "$MODEL" \
    --device 0 \
    --eval_recomp_ratio 0.1 \
    --eval_recomp_method "$M"
done
```

Cache ratio (Table 3). Repeat the same attack/evaluation workflow with $\delta \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and `-gcg_recomp_ratio 0.1`:

```
DATASET=.data/datasets/hotpotqa_200.json
for D in 0.1 0.2 0.3 0.4 0.5; do
  uv run python src/run_attack.py \
    --dataset "$DATASET" \
    --model "$MODEL" \
    --device 0,1,2,3 \
    --max_samples 200 \
    --cache_ratio "$D" \
    --chunk_size 32 \
    --gcg_recomp_method vanilla \
    --gcg_recomp_ratio 0.1 \
    --eval_recomp_method vanilla \
    --eval_recomp_ratio 0.1 \
    --eval_batch_size 32
done
```

Then use the preceding `run_evaluate.py` loop for each generated experiment directory. The cache ratio is the proportion of the context occupied by matched chunks; it is distinct from the recomputation ratio.

Recomputation ratio (Table 4). For each method and $\rho \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, optimize and score with the same method and ratio:

```
DATASET=.data/datasets/hotpotqa_200.json
for M in random epic cacheblend; do
  for R in 0.1 0.2 0.3 0.4 0.5; do
    uv run python src/run_attack.py \
      --dataset "$DATASET" \
      --model "$MODEL" \
      --device 0,1,2,3 \
      --max_samples 200 \
      --cache_ratio 0.3 \
      --chunk_size 32 \
      --gcg_recomp_method "$M" \
      --gcg_recomp_ratio "$R" \
      --eval_recomp_method "$M" \
      --eval_recomp_ratio "$R" \
      --eval_batch_size 32
  done
done
```

The saved experiment directory printed by each run is the input to `run_evaluate.py`. Because each full run is expensive, use the four-sample basic test to validate model/backend compatibility first. The resulting sweep metrics support C2.

(E4) RQ3 / Table 5: multi-turn persistence [30 human-minutes + approximately 1 compute-hour, depending on the target model]. RQ3 uses HotpotQA and keeps the optimized prefix fixed while inserting unrelated passages from other HotpotQA examples between the cached context and the target query. Construct five JSON variants with filler lengths $L_{\text{context}} \in \{0, 256, 512, 1024, 2048\}$ model tokens. Preserve the original `id`, `question`, `answer`, and `target`; insert the filler only at the context/query boundary and preserve each record's `gcg_prefix_ids` from the base attack result. Re-evaluate each variant with `run_evaluate.py` and compare the metrics across filler lengths; these results support C3.

(E5) RQ4 / Table 6: black-box transferability [20 human-minutes + about 12.5 compute-hours for proxy optimization, plus target evaluation]. First optimize the prefixes on the Llama-3.1-8B proxy model:

```
uv run python src/run_attack.py \
  --dataset .data/datasets/hotpotqa_200.json \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --device 0,1,2,3 \
  --max_samples 200 \
  --cache_ratio 0.3 \
  --chunk_size 32 \
  --gcg_recomp_method vanilla \
  --gcg_recomp_ratio 0.1 \
  --eval_recomp_method vanilla \
  --eval_recomp_ratio 0.1 \
  --eval_batch_size 32
```

Let `<EXPERIMENT_DIR>` be the resulting directory. Re-evaluate the same saved prefixes with each target model:

```
for TARGET in \
  Qwen/Qwen3-4B \
  Qwen/Qwen3-8B \
  Qwen/Qwen3-14B \
  meta-llama/Llama-3.2-1B \
  meta-llama/Llama-3.2-3B \
  meta-llama/Llama-3.3-70B; do
  uv run python src/run_evaluate.py \
    --dataset <EXPERIMENT_DIR> \
    --model "$TARGET" \
    --device 0 \
    --eval_recomp_ratio 0.1 \
    --eval_recomp_method vanilla
done
```

Repeat the evaluation loop for the other three recomputation methods. When the proxy and target tokenizers differ, the evaluator records the target-model metrics using its cross-tokenizer path; this is the intended black-box setting. Large target models require a correspondingly large-memory GPU. The resulting cross-model metrics support C4.

A.5 Notes on Reusability

The attack and evaluation interfaces accept any JSON list, or an object with a top-level `result` list, whose records provide `context`, `question`, `answer`, and `target`. The model is selected through `-model`; the cache-system controls are `-chunk_size`, `-cache_ratio`, `-gcg_recomp_*`, and `-eval_recomp_*`. The GCG budget can be varied with `-prefix_chunk_num`, `-num_steps`, `-topk`, `-search_width`, and `-eval_batch_size`.

The attack performs optimization locally and does not require access to a victim cache, victim prompts, or model weights beyond the locally selected surrogate. The resulting prefixes can be saved and re-evaluated under a different model or cache-recomputation method, which supports adaptive-attack, transferability, and defense-oriented studies beyond the paper's default settings.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.