ACCELERATING SUFFIX JAILBREAK ATTACKS WITH PREFIX-SHARED KV CACHE

Anonymous authors
Paper under double-blind review

ABSTRACT

Suffix jailbreak attacks are a systematic tool for red-teaming large language models (LLMs), but they require evaluating many candidate suffixes before finding an effective jailbreak. This paper presents Prefix-Shared KV Cache (PSKV), a plug-and-play inference optimization tailored to suffix jailbreak generation. PSKV exploits the structure that many candidate prompts share the same harmful instruction prefix while differing only in the candidate suffix. Instead of redundantly recomputing or physically duplicating the prefix cache, PSKV stores one compact prefix KV cache and expands it lazily at each layer during candidate evaluation. This design preserves PyTorch autograd compatibility and enables larger batched searches with lower memory overhead. Across six suffix attacks and five open-source LLMs, PSKV reduces inference time by about 40% and peak memory by about 50% while preserving attack effectiveness within the stochastic variation of the underlying attacks.

1 INTRODUCTION

Suffix jailbreak attacks, which aim to synthesize jailbreak suffixes to bypass safety guardrails and elicit harmful content, have become a vital component for red-teaming Large Language Models (LLMs) (Zou et al., 2023; Chao et al., 2024). Current approaches primarily employ heuristic search (Zou et al., 2023; Zhu et al., 2023) or LLM-based generators (Paulus et al., 2025) to generate these suffixes. However, a major bottleneck of these methods is their prohibitively high computational cost. Standard optimization-based attacks must iteratively evaluate a vast number of *candidate* suffixes against the target LLM to identify effective jailbreaks. While LLM-based generators offer faster test-time suffix generations, they merely shift the computational burden to the training phase, which still necessitates time-consuming suffix generation to construct training data.

Ideally, Key-Value (KV) caching could accelerate this evaluation process by reusing intermediate tensors. However, in jailbreak scenarios, the system prompt is typically far longer than the suffix. Standard KV cache implementations must duplicate this long prompt's cache for every candidate suffix in a batch, inducing excessive memory consumption that limits batch size. While state-of-the-art inference systems like vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) address these memory constraints through careful memory scheduling, they are designed strictly for forward-pass inference. Their highly encapsulated interfaces and lack of backpropagation kernels prevent access to gradients, restricting their utility to a subset of gradient-free attacks (*e.g.*, BEAST attack (Sadasivan et al., 2024)). Consequently, these limitations collectively impede the effective acceleration of general suffix jailbreak attacks.

To overcome these limitations, we propose **Prefix-Shared KV Cache (PSKV)**, a lightweight, workload-aware, and plug-and-play framework tailored for accelerating suffix jailbreak attacks. Our method is motivated by a key *structural redundancy* in jailbreak optimization: while a batch of candidate suffixes varies, they typically share an identical targeted harmful instruction as the prefix. Standard KV cache implementations ignore this property, redundantly duplicating the long prefix cache for every candidate. In contrast, PSKV maintains a single, shared KV cache for the invariant prefix and broadcasts it to all candidate suffixes during inference. By employing a suffix-centric alignment strategy, PSKV supports dynamic expansion and parallel batched loss computation. Unlike encapsulated inference engines, PSKV integrates seamlessly into existing attack pipelines, ensuring substantial acceleration and minimal memory overhead while preserving gradient access.

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.

We evaluated PSKV across six widely used suffix jailbreak attacks and five popular open-source models. We compared our framework against a baseline without KV cache and a standard KV cache implementation. The results demonstrate that PSKV reduces inference time by approximately 40% compared to the no-cache baseline and decreases peak memory usage by 50% compared to the standard KV cache implementation. Furthermore, because PSKV is a drop-in optimization that does not alter the high-level logic of the attack, it maintains the original Attack Success Rate (ASR). This efficiency enables researchers to scale evaluations to larger models and broader search spaces that were previously computationally prohibitive. Additional analyses in Appendix E clarify PSKV's distinction from serving engines, its plug-and-play integration path, ASR stability, and responsible release considerations.In summary, our contributions are:

- We identify the inefficiency of general-purpose inference caching when applied to the specific structural patterns of jailbreak suffix optimization.
- We propose PSKV, a memory-efficient framework featuring layer-wise lazy expansion and suffix-centric tensor alignment to eliminate redundant prefix computation while preserving autograd support.
- We conduct comprehensive evaluations demonstrating PSKV substantially reduces both computational and memory costs, paving the way for scalable LLM vulnerability assessment.

2 Related Work

Automated Suffix Jailbreaking. Recent research has transitioned from manual red-teaming to automated adversarial optimization. The foundational method, GCG (Zou et al., 2023), employs a greedy, gradient-guided search to iteratively refine an adversarial suffix. Building on this, subsequent works have explored more sophisticated search strategies. BEAST (Sadasivan et al., 2024) replaces the greedy algorithm with beam search to explore a wider range of promising candidates. Zhu's AutoDAN (Zhu et al., 2023) utilizes genetic algorithms to evolve suffixes, mitigating the local optima problem of purely greedy methods. Other variants like GCQ (Gradient-based Attack with Quantization) (Hayase et al., 2024) focus on improving the efficiency of the core gradient-based update step itself.

An alternative paradigm uses LLMs themselves as generators. AdvPrompter (Paulus et al., 2025) fine-tunes a language model to automatically compose adversarial prompts. Similarly, AmpleGCG trains a generative model to directly output effective suffixes, amortizing the search cost into a one-time training process (Liao & Sun, 2024). Although these methods differ, they all require evaluating a large number of candidate suffixes to create a training set during either the model training or data generation phases. Our work presents an orthogonal optimization that accelerates this critical evaluation step, benefiting both categories of attack.

Efficient Inference & KV Caching. Optimizing Key-Value (KV) cache management is central to high-throughput LLM serving. vLLM (Kwon et al., 2023) introduced PagedAttention to mitigate memory fragmentation in multi-user scenarios, while SGLang (Zheng et al., 2024) employs Radix-Attention to automatically detect and share prefixes across diverse incoming requests via a radix tree structure. ChunkAttention (Ye et al., 2024) further optimizes memory locality. However, these systems are architected for *asynchronous, multi-user serving*, prioritizing request scheduling and fairness over raw batch latency. Applying them to the *synchronous, tight-loop* nature of adversarial attacks introduces unnecessary overhead (e.g., complex tree lookups, scheduler latency). In contrast, PSKV is a workload-specific optimization that hard-codes the "one-prefix-many-suffixes" pattern directly into the inference kernel, bypassing the complexity of general-purpose serving engines to achieve maximum efficiency for attack loops.

3 Preliminaries

As illustrated in Fig 1, the workflow of a typical suffix-based jailbreak attack operates as an iterative cycle comprising three phases: heuristic candidate generation, forward pass evaluation, and candidate selection. This pipeline forms the operational basis for the optimization problem defined below.

Adversarial Suffix Optimization. We consider an LLM as a conditional distribution $p_\theta(y|x)$ over a vocabulary $\mathcal{V}$. Given a harmful query $x^{(h)}$ and a target response $y^{(h)}$, the goal of a suffix attack is to find an adversarial sequence $x^{(s)} \in \mathcal{V}^L$ that maximizes the target probability:

$$x^{(s)*} = \arg\max_{x^{(s)} \in \mathcal{V}^L} \log p_\theta(y^{(h)}|x^{(h)} \oplus x^{(s)}), \quad (1)$$

where $\oplus$ denotes concatenation. Solving Eq. (1) typically involves iterative discrete optimization. In the *heuristic search* phase, algorithms mutate the current suffix to generate a batch of N candidates, $\mathcal{S} = \{s_1, \ldots, s_N\}$. Subsequently, during the *response generation* phase, these candidates are concatenated with the static prompt and the target response to form the batch $\{x^{(h)} \oplus s_i \oplus y^{(h)}\}_{i=1}^N$. This batch is then fed into the model to compute the prediction logits for the target sequence. Based on the computed target and perplexity losses, the top-k candidates are *selected* in the suffix selection phase to seed the next iteration. Crucially, this cycle repeats for hundreds of epochs, meaning the static prefix $x^{(h)}$ is re-evaluated N times per step, creating a massive computational workload.

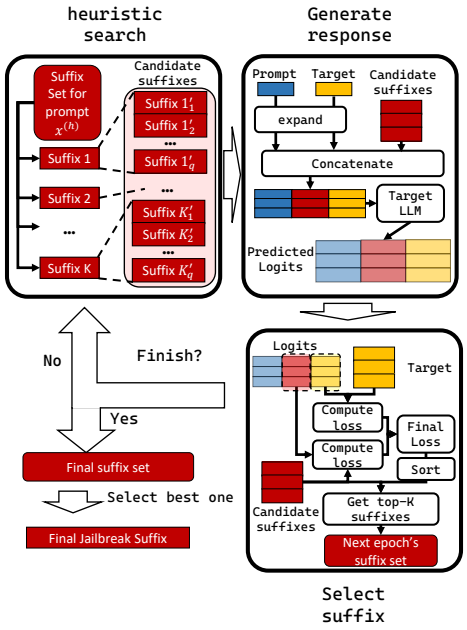

Figure 1: The computational pipeline of iterative suffix attacks.

Standard Attention and KV Cache. The core operator in Transformer-based LLMs (Vaswani et al., 2017) is the scaled dot-product attention. For a specific layer and attention head, given a query q_t, and the key-value pairs for the sequence history $(K_{<t}, V_{<t})$, the attention output o_t is computed as:

$$o_t = \text{Softmax}\left(\frac{q_t[K_{<t}; k_t]^\top}{\sqrt{d}}\right)[V_{<t}; v_t], \quad (2)$$

where $[\cdot;\cdot]$ denotes concatenation along the sequence length dimension. To avoid recomputing $K_{<t}$ and $V_{<t}$ at every step, inference systems employ KV Caching to store these tensors in GPU memory. Since standard deep learning frameworks require the KV cache batch dimension to align with the input batch size N, the KV states for the static prefix $x^{(h)}$ must be physically duplicated N times:

$$\mathcal{M}_{prefix} = \text{Repeat}(K^{(h)}, V^{(h)}, \text{times} = N). \quad (3)$$

This results in a memory complexity of $\mathcal{O}(N \cdot L_{prefix})$, which strictly limits the search width N and prevents the use of large batch sizes.

4 Methodology

In this section, we first illustrate the redundant computation problem of prefix KV vectors in suffix jailbreaking and then introduce our *PSKV* method to tackle the raised challenge and substantially improve the efficiency of suffix jailbreak attacks.

4.1 Redundant Prefix computation in Suffix Jailbreak Attacks

According to the general jailbreak suffix synthesis process described in Section 3, the main computational burden of suffix jailbreaking arises from performing $K \cdot q$ inference passes (in the response generation phase) in each synthesis iteration, where K is the number of candidate suffixes in each iteration, and q is the number of new suffixes generated for each candidate. Larger K and q usually lead jailbreak attacks to explore more candidate suffixes, thus increasing the chance of finding an effective jailbreak suffix; however, they also result in more LLM inference times.

The most straightforward implementation, which we term the **unaccelerated traditional attack method**, treats each of the $K \cdot q$ candidate prompts as independent inputs in every iteration. This

approach repeatedly re-computes the KV vectors for the prefix $x^{(h)}$, leading to significant redundant computational overhead. As a result, when an LLM is performing inference on these prompts in parallel, it will redundantly calculate the same KV vectors for the same prefix $x^{(h)}$ for up to $K \cdot q$ times at each step.

A more advanced baseline would employ **standard KV cache**. The KV vectors for the prefix $x^{(h)}$ are computed only once before the iterative synthesis process begins. However, to evaluate all $K \cdot q$ suffix candidates in a single parallel batch, this pre-computed prefix cache must be duplicated $K \cdot q$ times to match the batch dimension of the suffixes, which introduces a severe memory bottleneck. For a batch of B prompts, this problem is exacerbated, requiring $B \cdot K \cdot q$ copies of the caches.

Furthermore, as shown in Table 1, the average token length of instructions in HarmBench is 78.4 (Mazeika et al., 2024), which is significantly larger than the standard jailbreak suffix length in practice (typically 20 tokens). These observations suggest that if the KV vectors of the prefix harmful instruction $x^{(h)}$ can be calculated only once and used for the inference of different candidate prompts, then both the *computation* and *memory* costs of suffix jailbreaking can be considerably reduced, as a large amount of redundant intermediate KV vectors no longer need to be calculated.

Fortunately, we observe that while the number of candidate prompts an LLM needs to perform inference on in each jailbreaking iteration is as large as $K \cdot q$, these prompts share the same prefix, *i.e.*, the original harmful instruction $x^{(h)}$. As illustrated in Fig. 1, suffix jailbreak attack only modifies the suffixes of these candidate prompts but keeps the harmful instruction $x^{(h)}$ as their prefix intact.

4.2 PSKV FOR SINGLE JAILBREAK SUFFIXES SYNTHESIS

These observations highlight a clear opportunity for optimization. Based on our previous analysis, we now propose *PSKV*, a method that can substantially accelerate existing suffix jailbreak attacks by reducing redundant computation of prefix KV vectors during the candidate prompt inference process of suffix jailbreaking.

Table 1: The average length (in tokens) of harmful prompts and targeted texts in HarmBench.

Dataset	Harmful Prompt	Targeted Text
HarmBench	78.4	93.3

The overall procedures are also illustrated in Figure 2. PSKV adopts a lazy, layer-wise expansion strategy. Unlike standard implementations that pre-allocate expanded memory for all layers during initialization, we maintain a compact, single-batch storage of the prefix. Specifically, before a suffix jailbreak attack starts the iterative suffix searching process in Figure. 1, PSKV runs a single forward pass for the harmful instruction $x^{(h)}$ to compute and cache its KV vectors across all attention layers of the targeted LLM. Then, when the targeted LLM is performing inference on the $K \cdot q$ candidate prompts during the iterative suffix searching process, whenever the LLM needs to use the KV vectors of the prefix from an attention layer, PSKV directly retrieves the corresponding KV vectors from its cache and duplicates these vectors $K \cdot q$ times for the LLM to support its inference process. Compared with vanilla suffix jailbreaking, our PSKV method reduces the time-consuming prefix KV cache computation from $K \cdot q$ times to only **a single time**, which can significantly improve the computation efficiency of suffix jailbreaking.

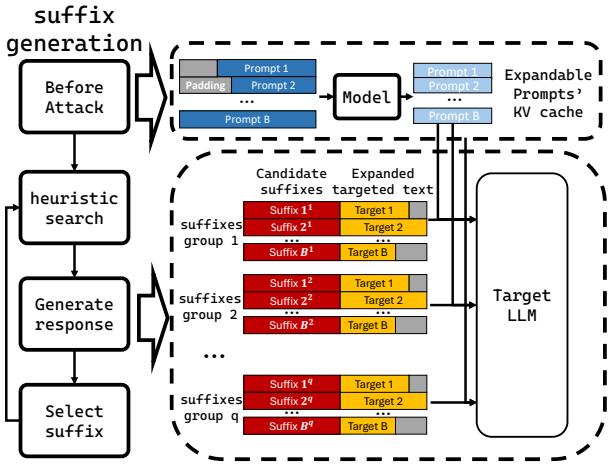

Figure 2: PSKV. By reusing the prefix KV cache, we only need to store one copy of the prompt for q new suffixes, rather than q copies as required by traditional methods.

Table 2: Unified complexity analysis. E: epochs; N_{cand}: total candidates per iteration ($K \cdot q$ for single, $B \cdot K \cdot q$ for batch); N_p, N_s, N_t: lengths of prefix, suffix, and target. $L_{dec} = N_s + N_t$ denotes decoding length.

Scenario	Method	Computational Complexity	Peak Memory Complexity
Single Instruction	No Accel.	$\mathcal{O}(E \cdot N_{cand} \cdot (N_p + L_{dec})^2)$	$\mathcal{O}(N_{cand} \cdot (N_p + L_{dec}))$
	Standard KV	$\mathcal{O}(N_p^2 + E \cdot N_{cand} \cdot L_{dec} \cdot (2N_p + L_{dec}))$	$\mathcal{O}(N_{cand} \cdot (\mathbf{N_p} + L_{dec}))$
	PSKV	$\mathcal{O}(N_p^2 + E \cdot N_{cand} \cdot L_{dec} \cdot (2N_p + L_{dec}))$	$\mathcal{O}(\mathbf{1} \cdot N_p + N_{cand} \cdot L_{dec})$
Batched Multi-Instruction	No Accel.	$\mathcal{O}(E \cdot N_{cand} \cdot (N_{p,\text{avg}} + L_{dec})^2)$	$\mathcal{O}(N_{cand} \cdot (N_{p,\text{avg}} + L_{dec}))$
	Standard KV	$\mathcal{O}(B \cdot N_p^2 + E \cdot N_{cand} \cdot L_{dec} \cdot (2N_p + L_{dec}))$	$\mathcal{O}(N_{cand} \cdot (\mathbf{N_{p,\text{max}}} + L_{dec}))$
	PSKV	$\mathcal{O}(B \cdot N_p^2 + E \cdot N_{cand} \cdot L_{dec} \cdot (2N_p + L_{dec}))$	$\mathcal{O}(\mathbf{B} \cdot N_{p,\text{max}} + N_{cand} \cdot L_{dec})$

Furthermore, we note that the KV vector duplicating procedure in PSKV requires little memory overhead. This is because during LLM inference, the outputs of all LLM attention layers are calculated in a layer-by-layer manner, and the computation process only requires the prefix KV vectors from that specific layer. As a result, the prefix KV vector duplication procedure in PSKV is also performed in a layer-by-layer manner, and at any given time, only KV vectors from a single layer are duplicated. This ensures that at any given moment, the memory-intensive expansion is isolated to a single layer, considerably reducing peak memory overhead compared to the vanilla approach where redundant prefix repetitions persist across the entire network stack simultaneously.

Complexity analysis. Let E be the number of synthesis iterations; N_p, N_s, N_t the token lengths of the prefix, suffix, and target; and $K \cdot q$ the candidates per iteration. We compare the total computational (C) and memory (M) complexities. We provide the detailed proof in Appendix B.1.

Table 2 summarizes the complexity analysis (detailed proof in Appendix B.1). While Standard KV caching reduces computational cost, it incurs a high memory footprint of $\mathcal{O}(N_{cand} \cdot N_p)$ due to prefix duplication. In contrast, PSKV maintains the same computational efficiency but reduces the memory complexity, effectively removing the dependency of prefix memory on the batch size. Appendix B.3 analyzes how prefix, suffix, and target lengths, search width, micro-batching, and model dimensions affect compute and memory costs.

4.3 Suffix-Centric Alignment for Batched Multi-Instruction Attack

In real-world benchmarking, evaluating robust accuracy requires optimizing suffixes for a batch of B distinct instruction-response pairs $\{(x_i^{(h)}, y_i^{(h)})\}_{i=1}^{B}$. A practical challenge arises from the variable token lengths of these pairs, which typically leads to ragged tensors that hinder efficient parallelization. To resolve this and maximize the utility of PSKV, we implement a **Suffix-Centric Alignment** strategy.

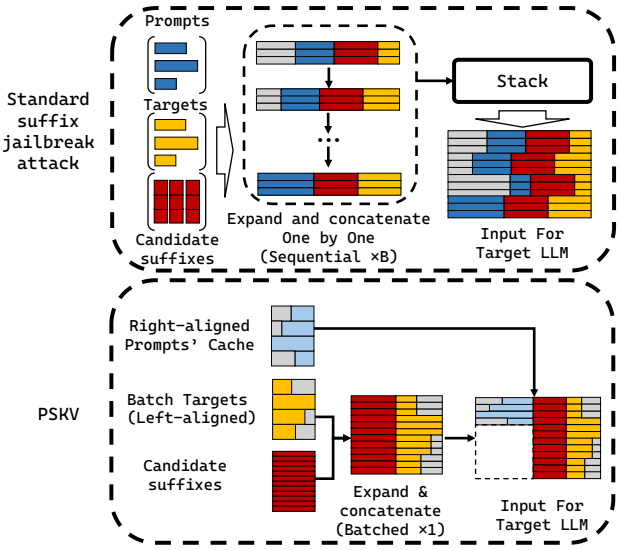

Figure 3: Alignment strategies for batched suffix attacks. The standard method's misaligned structure prevents prompt KV cache reuse. PSKV utilizes a suffix-centric alignment, allows all components to be processed in a single, highly efficient tensor operation.

As illustrated in Figure 3, we normalize the data layout by left-padding all instructions $x_i^{(h)}$ and right-padding all target responses $y_i^{(h)}$. This alignment ensures that all generated suffixes start at the same column index within the batch tensor. Consequently, despite the variable input lengths, the suffixes form a single, dense token matrix of shape $(B \cdot K \cdot q) \times N_s$.

Table 3: Attack Success Rates (ASRs, %) of suffix jailbreak attacks with and without PSKV. Values are mean ± standard deviation over three runs for each attack/model pair.

Model	Cache	GCG	GCQ	AutoDAN	BEAST	AmpleGCG	AdvPrompter
Vicuna-7B	None	98.0 ± 2.0	90.0 ± 3.5	54.7 ± 3.1	94.0 ± 2.0	94.7 ± 3.1	97.3 ± 1.2
	PSKV	96.0 ± 2.0	93.3 ± 2.3	53.3 ± 5.0	95.3 ± 4.2	93.3 ± 2.3	96.0 ± 2.0
Llama-2-7B	None	64.0 ± 3.5	18.0 ± 0.0	24.7 ± 1.2	39.3 ± 3.1	93.3 ± 2.3	91.3 ± 4.2
	PSKV	66.7 ± 1.2	17.3 ± 1.2	25.3 ± 3.1	42.7 ± 3.1	92.7 ± 4.2	92.0 ± 5.3
Llama-3-8B	None	62.7 ± 2.3	25.3 ± 1.2	28.0 ± 4.0	48.7 ± 3.1	70.7 ± 2.3	74.7 ± 3.1
	PSKV	65.3 ± 9.5	26.0 ± 0.0	28.0 ± 2.0	50.7 ± 4.2	72.0 ± 3.5	77.3 ± 4.2
Mistral-7B	None	93.3 ± 4.2	91.3 ± 3.1	82.7 ± 4.2	85.3 ± 3.1	86.0 ± 4.0	92.7 ± 2.3
	PSKV	94.0 ± 4.0	92.0 ± 5.3	81.3 ± 3.1	86.7 ± 3.1	88.7 ± 4.2	93.3 ± 2.3
Qwen-2.5-7B	None	86.0 ± 2.0	86.7 ± 3.1	55.3 ± 4.2	89.3 ± 3.1	82.0 ± 3.5	92.7 ± 4.2
	PSKV	89.3 ± 2.3	89.3 ± 2.3	55.3 ± 5.0	87.3 ± 1.2	86.7 ± 2.3	93.3 ± 1.2

This design yields two critical benefits. The dense matrix representation allows the massive number of candidate suffixes ($B \cdot K \cdot q$) to be processed using highly optimized vectorized operations in standard frameworks (e.g., PyTorch), eliminating the computational overhead from sequential or irregular batching in loss evaluation and suffix retrieval. Meanwhile, the right-alignment of instructions aligns the memory layout of the static prefixes. This allows PSKV to pre-compute the KV cache once for the B instructions and broadcast it efficiently across the massive candidate dimension.

Complexity analysis (batch). We analyze the complexity for a batch of B distinct prompts, generating a total of $N_{cand} = B \cdot K \cdot q$ candidates per iteration (detailed proofs in Appendix B.2). As summarized in Table 2, the **Standard KV cache** substantially reduces computational costs compared to the unaccelerated baseline, but it requires physically duplicating the prefix KV states for every candidate to match the input batch dimension, resulting in $\mathcal{O}(N_{cand} \cdot N_{p,\text{max}})$ prefix memory footprint.

In contrast, by sharing the KV cache across all suffixes targeting the same prompt, PSKV reduces the prefix memory term to $\mathcal{O}(B \cdot N_{p,\text{max}})$. This term scales with the number of prompts B, not the much larger total candidate count N_{cand} (since typically $K \cdot q \gg 1$). This optimization effectively removes the memory barrier, making large-scale parallel attacks practical. Appendix E.4 describes the extension to shared universal suffixes and reports a controlled Universal GCG experiment.

5 Experiments

In this section, we empirically explore how our proposed PSKV method can reduce the computation and memory costs of suffix jailbreakings while preserving their attack effectiveness.

5.1 Experimental Setup

Safety dataset. We use the first 50 examples from the test set of **HarmBench** (Mazeika et al., 2024) to form the safety dataset for the suffix jailbreaking evaluations. Each sample from this safety dataset consists of a harmful instruction and a targeted harmful response.

Targeted Models. We empirically validated PSKV using six suffix attacks on five target LLMs, which are: **Vicuna-7B-v1.5** (Chiang et al., 2023), **Llama-2-7B-Chat** (Touvron et al., 2023), **Llama-3-8B-Instruct** (Grattafiori et al., 2024), **Mistral-7B-Instruct-v0.3** (Jiang et al., 2023), and **Qwen2.5-7B-Instruct** (Qwen et al., 2025). Appendix E.2 reports additional GCG experiments on Llama-2-13B and larger HarmBench evaluation sets.

Suffix jailbreak attacks. We benchmark how our PSKV can improve the efficiency of conducting six common suffix jailbreak attacks from two families. Specifically, **GCG** (Zou et al., 2023), **GCQ** (Hayase et al., 2024), **Zhu's AutoDAN** (Zhu et al., 2023)(abbreviated as **AutoDAN** hereafter), and **BEAST** (Sadasivan et al., 2024) are from the family of optimization-based attacks, which aim to search for jailbreak suffixes using various discrete optimization techniques. **AmpleGCG** (Liao & Sun, 2024) and **AdvPrompter** (Paulus et al., 2025) are from the family of model-based attacks, which aim to train LLM-based generators to directly generate jailbreak suffixes for given harmful

Table 4: Time usage (min) for (1) optimization-based suffix synthesis and (2) model-based training sample generation. Parentheses (×) indicate speed-up factors vs. the unaccelerated baseline. **OOM**: Out Of Memory.

Model	Acceleration	Optimization-based Attacks				Model-based Attacks	
		GCG	GCQ	AutoDAN	BEAST	AmpleGCG	AdvPrompter
Vicuna-7B	None	228 (× 1.00)	247 (× 1.00)	88 (× 1.00)	23 (× 1.00)	349 (× 1.00)	462 (× 1.00)
	Standard KV	147 (× 1.55)	OOM	OOM	OOM	239 (× 1.46)	160 (× 2.89)
	PSKV (Ours)	146 (× 1.56)	148 (× 1.67)	50 (× 1.76)	15 (× 1.53)	253 (× 1.38)	199 (× 2.32)
Llama-2-7B	None	221 (× 1.00)	210 (× 1.00)	86 (× 1.00)	26 (× 1.00)	349 (× 1.00)	513 (× 1.00)
	Standard KV	142 (× 1.56)	OOM	OOM	OOM	242 (× 1.44)	344 (× 1.49)
	PSKV (Ours)	142 (× 1.56)	145 (× 1.45)	48 (× 1.79)	14 (× 1.86)	227 (× 1.54)	275 (× 1.87)
Llama-3-8B	None	288 (× 1.00)	221 (× 1.00)	88 (× 1.00)	24 (× 1.00)	329 (× 1.00)	577 (× 1.00)
	Standard KV	151 (× 1.91)	141 (× 1.57)	48 (× 1.83)	OOM	240 (× 1.37)	282 (× 2.05)
	PSKV (Ours)	141 (× 2.04)	138 (× 1.60)	45 (× 1.96)	14 (× 1.71)	242 (× 1.36)	307 (× 1.88)
Mistral-7B	None	211 (× 1.00)	203 (× 1.00)	81 (× 1.00)	25 (× 1.00)	320 (× 1.00)	96 (× 1.00)
	Standard KV	138 (× 1.53)	126 (× 1.61)	40 (× 2.03)	OOM	229 (× 1.40)	80 (× 1.20)
	PSKV (Ours)	131 (× 1.61)	122 (× 1.66)	40 (× 2.03)	14 (× 1.79)	230 (× 1.39)	74 (× 1.30)
Qwen2.5-7B	None	250 (× 1.00)	242 (× 1.00)	84 (× 1.00)	26 (× 1.00)	295 (× 1.00)	102 (× 1.00)
	Standard KV	134 (× 1.87)	143 (× 1.69)	44 (× 1.91)	OOM	223 (× 1.32)	81 (× 1.26)
	PSKV (Ours)	135 (× 1.85)	128 (× 1.89)	42 (× 2.00)	15 (× 1.73)	237 (× 1.24)	76 (× 1.34)

instructions. Please refer to Appendix A for detailed settings of all other hyperparameters of all attacks. All of the methods were benchmarked on 2 80GB A100 GPUs.

Evaluation metrics. For the attack performance evaluation, we report the **Attack Success Rate (ASR)**, defined as the percentage of successful jailbreak attacks on the safety dataset. Please refer to Appendix A for details. For attack efficiency evaluation, we adopt memory consumption in gigabytes (GB) and execution time in minutes as the metrics.

Baseline acceleration methods. Our experiments evaluate the effectiveness of our proposed *PSKV* by comparing it against two alternatives: a baseline implementation use **no specific acceleration method** (which we denote as **None**), and an implementation using a **Standard KV Cache**.

None represents an unaccelerated, albeit batched, approach. In each iteration, it constructs $K \cdot q$ full candidate prompts by concatenating the prefix $x^{(h)}$ with each suffix. It then performs a single, large batched forward pass on these $K \cdot q$ sequences. It redundantly calculates the prefix KV vectors for every candidate in every iteration. **Standard KV Cache** implements the intuitive optimization of pre-computing the prefix cache. It first performs a single forward pass on the prefix $x^{(h)}$ to obtain its KV cache. Then, to evaluate the $K \cdot q$ suffix candidates in a parallel batch, it physically replicates this prefix cache $K \cdot q$ times in GPU memory to match the batch dimension of the suffixes.

5.2 Results

PSKV does not affect the attack performance. According to the design of PSKV described in Section 4, PSKV is a performance-neutral acceleration method that does not affect the attack performance of accelerated suffix jailbreak attacks. This is because PSKV only removes redundant key-value vectors and does not change the algorithm itself or the computational accuracy. Table 3 reports ASR means and standard deviations over three runs.

Additional multi-seed checks in Appendix E.1 suggest that observed ASR differences are within the stochastic variation of the attacks. These variations stem not from PSKV altering the core attack logic but from inherent randomness in heuristic search, candidate sampling, beam search, and sampled response generation for ASR computation.

PSKV considerably accelerates suffix jailbreaking. Table 4 details the time consumption of varying acceleration strategies. First, for optimization-based attacks that perform only one or two forward passes per candidate suffix step (including GCG, GCQ, BEAST, and AmpleGCG), PSKV consistently achieves a speed-up ratio of approximately $1.4\times$ to $1.6\times$ compared to the no-acceleration baseline. Notably, this acceleration becomes even more pronounced for Zhu's AutoDAN, where the speed-up ratio rises to a range of $\sim 1.8\times$ to $1.95\times$ (e.g., reducing time from 88 to 45 minutes on Llama-3-8B). This performance gain is attributed to the mechanism of KV cache acceleration: it reduces computational redundancy by storing states. Since Zhu's AutoDAN requires substantially more

Table 5: Memory consumption (GB) for (1) optimization-based suffix synthesis and (2) model-based training sample generation. **OOM**: Out Of Memory. (×): Reduction factor vs. Standard KV Cache; (−) denotes baseline OOM.

Model	Acceleration	Optimization-based Attacks				Model-based Attacks	
		GCG	GCQ	AutoDAN	BEAST	AmpleGCG	AdvPrompter
Vicuna-7B	None	30 (× 3.35)	67 (-)	90 (-)	144 (-)	146 (× 1.02)	125 (× 1.25)
	Standard KV	100 (× 1.00)	OOM	OOM	OOM	148 (× 1.00)	155 (× 1.00)
	PSKV (Ours)	27 (× 3.68)	56 (-)	53 (-)	92 (-)	63 (× 2.36)	67 (× 2.32)
Llama-2-7B	None	35 (× 2.84)	63 (-)	102 (-)	112 (-)	145 (× 1.02)	73 (× 2.09)
	Standard KV	100 (× 1.00)	OOM	OOM	OOM	148 (× 1.00)	152 (× 1.00)
	PSKV (Ours)	30 (× 3.39)	50 (-)	51 (-)	81 (-)	58 (× 2.55)	63 (× 2.40)
Llama-3-8B	None	56 (× 1.05)	118 (× 1.06)	124 (× 1.10)	142 (-)	128 (× 0.88)	117 (× 1.27)
	Standard KV	58 (× 1.00)	125 (× 1.00)	136 (× 1.00)	OOM	112 (× 1.00)	149 (× 1.00)
	PSKV (Ours)	41 (× 1.43)	85 (× 1.47)	77 (× 1.77)	110 (-)	91 (× 1.24)	126 (× 1.18)
Mistral-7B	None	34 (× 1.33)	54 (× 1.56)	85 (× 1.35)	144 (-)	86 (× 0.93)	54 (× 1.69)
	Standard KV	46 (× 1.00)	84 (× 1.00)	114 (× 1.00)	OOM	80 (× 1.00)	90 (× 1.00)
	PSKV (Ours)	28 (× 1.62)	44 (× 1.91)	44 (× 2.59)	120 (-)	58 (× 1.37)	66 (× 1.37)
Qwen2.5-7B	None	66 (× 0.81)	122 (× 0.94)	141 (× 0.89)	139 (-)	121 (× 0.93)	127 (× 1.15)
	Standard KV	54 (× 1.00)	115 (× 1.00)	125 (× 1.00)	OOM	112 (× 1.00)	146 (× 1.00)
	PSKV (Ours)	42 (× 1.28)	94 (× 1.22)	82 (× 1.53)	115 (-)	94 (× 1.19)	138 (× 1.06)

inference steps per iteration (due to its hierarchical genetic algorithm), there is a larger proportion of redundant prefix computation for PSKV to eliminate.

Conversely, for AdvPrompter, the speed-up ratio decreases to approximately $1.3\times$. This is because AdvPrompter involves synchronous reinforcement learning; the total time includes both inference and generator training. While PSKV accelerates the inference phase, the training overhead remains constant, diluting the overall proportion of time saved. Furthermore, compared to the Standard KV Cache, our method maintains a comparable speed without incurring time penalties (in some cases, such as Llama-3 AdvPrompter, it is even faster). This indicates that the overhead caused by our temporary KV tensor expansion is negligible relative to the total inference time.

Finally, observing across different target models, the variance in time usage is largely within the range of random fluctuations.[1] Since the sequence lengths are relatively short (avg. $\sim$190 tokens), the computational bottleneck lies in computation rather than model's throughput limitations.

Memory Efficiency Analysis of PSKV. Table 5 reveals that PSKV consistently achieves superior memory efficiency, often surpassing even the "None" baseline by avoiding the storage of expanded input tensors required for batch processing. Unlike Standard KV Cache, which suffers from memory explosion under large beam widths by maintaining a fully expanded state, PSKV leverages layer-wise inference to dynamically expand and immediately discard the KV view for the current layer, keeping the resident footprint minimal.

Memory savings vary by attack dynamics: for gradient-based attacks (GCG) where the KV cache dominates memory usage, PSKV achieves high compression; conversely, while massive intermediate tensors in beam-search methods (GCQ, BEAST) lower the relative reduction ratio, PSKV remains vital for preventing the OOM errors frequent in Standard Cache. Similarly, PSKV mitigates cumulative cache growth in AutoDAN, though in dual-model setups like AdvPrompter, the increased non-KV memory burden can marginally offset these savings.

Architecturally, the efficiency gap is most pronounced with Multi-Head Attention (MHA) models (Vicuna-7B, Llama-2-7B) where PSKV eliminates the redundancy of storing unique pairs for every head.[2] In contrast, models utilizing Grouped-Query Attention (GQA) like Llama-3, Mistral,[3] and Qwen2.5 inherently reduce the baseline KV size, lowering PSKV's theoretical compression ceiling. Furthermore, the larger vocabularies in these models generate massive intermediate tensors (logits, IDs) that further dilute the KV cache's relative memory proportion, rendering relative savings less

[1] We also observe that AdvPrompter requires significantly more time on Llama-series models compared to Mistral and Qwen. For explanations, please refer to App. C.

[2] See Appendix D for architectural details.

[3] Mistral's sliding window does not impact these results as inputs fit within the window.

apparent. See Appendix D for a more detailed analysis of memory consumption. Appendix E.3 reports additional experiments with longer suffixes and WildJailbreak prompts.

Scalability Analysis. Table 6 highlights PSKV's efficient memory scalability. Even the total peak memory consumption remains virtually constant despite a 4× increase in search width. This stability arises because the peak memory footprint is dominated by substantial intermediate tensors required for gradient computation and loss computation, rather than the inference cache itself. Since PSKV enables efficient prefix sharing, the marginal memory cost of expanding the search width is negligible and remains entirely masked beneath the high-water mark set by these non-cache components. Consequently, the amortized memory cost per candidate drops inversely with the width, demonstrating that PSKV enables high-width parallel search without raising the hardware memory barrier.

Table 6: Scalability of iterative/search-based attacks with PSKV on HarmBench using a Llama-3-8B target.

Attack	Width	Total Cost		Cost/width		ASR (%)
		Time (min)	Mem. (GB)	Time (min)	Mem. (GB)	
GCG	16	40	40.83	2.50	2.55	50
	32	73	40.83	2.28	1.28	54
	64	141	40.84	2.20	0.64	62
GCQ	16	34	85.06	2.13	5.32	26
	32	70	85.07	2.19	2.66	26
	64	138	85.07	2.16	1.33	26
BEAST	5	5	110.27	1.00	22.05	40
	10	9	110.29	0.90	11.03	50
	15	14	110.29	0.93	7.35	54

Comparisons with State-of-the-Art Inference Systems. We include vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) in our evaluation to benchmark our method against comprehensively optimized inference systems used in production. While these frameworks utilize advanced memory management techniques like PagedAttention, they lack the gradient support required for white-box optimization (e.g., GCG, AdvPrompter). However, for the gradient-free **BEAST** attack, they serve as critical baselines. By comparing against these highly specialized engines, we demonstrate the computational superiority and efficiency of our implementation.

Table 7: Efficiency of gradient-free BEAST attack on specialized inference frameworks (SGLang and vLLM).

framework	Cost	model				
		vicuna	llama2	llama3	mistral	Qwen2.5
sglang	Time	8:49	9:25	6:22	7:00	6:12
	Mem.	13.42	12.87	14.96	13.50	14.23
vLLM	Time	16:02	16:50	17:58	16:27	15:51
	Mem.	12.55	12.57	14.96	12.73	13.52

Table 7 benchmarks the performance of the BEAST attack on specialized inference frameworks. Although vLLM and SGLang exhibit impressive efficiency through their full-stack optimized kernels, their design explicitly sacrifices gradient computation, making them unsuitable for white-box optimization tasks. PSKV-accelerated BEAST (see Tables 4 and 5) retains full automatic differentiation support within a native PyTorch environment, yielding performance that is only slightly inferior to—or in some configurations, even comparable to—these two industry-standard engines. Furthermore, while vLLM and SGLang are restricted to gradient-free attacks, PSKV serves as a universal plug-and-play accelerator across the entire spectrum of suffix jailbreak methodologies, bridging the gap between high-performance inference and flexible adversarial optimization.

6 Conclusion

The effectiveness of suffix jailbreak attacks is fundamentally tied to heuristic searches that require a vast number of resource-intensive LLM inferences, creating a significant barrier to efficient security evaluation. For a single harmful prompt, numerous candidate suffixes are generated, yet standard methods create multiple redundant copies of the prompt's computational state, leading to massive memory waste. To address this inefficiency, we propose the *Prefix-Shared KV Cache (PSKV)* framework. The core principle of PSKV is to maintain only a single, shared, and expandable KV cache for each harmful instruction prefix, which is then efficiently reused for all its corresponding suffix candidates. To enable this sharing mechanism at scale for batches of diverse prompts, we also introduce a suffix-centric alignment strategy. We empirically validated PSKV across six suffix attacks on five LLMs, demonstrating an average reduction of 40% in time and 50% in memory compared to cache-optimized baselines, while preserving the original attack success rate.

AI USE STATEMENT

Generative AI tools assisted with LaTeX template migration and formatting, organization of previously obtained experimental results in the appendix, and drafting and checking the accompanying complexity analysis against the implementation. The authors are responsible for the final content.

ETHICS STATEMENT

This work studies an acceleration technique for existing suffix jailbreak attacks and is intended to support scalable safety evaluation and red-team auditing. PSKV does not introduce a new attack objective, harmful dataset, or model vulnerability; it reduces redundant computation in already published attack pipelines. Because lower compute cost can also reduce the cost of misuse, any public release should be research-only, gated by an acceptable-use policy, and distributed as a modular acceleration library without bundled harmful prompt datasets or ready-to-run malicious attack scripts.

REFERENCES

Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J. Pappas, and Eric Wong. Jailbreaking black box large language models in twenty queries, 2024. URL https://arxiv.org/abs/2310.08419.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

Jonathan Hayase, Ema Borevkovic, Nicholas Carlini, Florian Tramèr, and Milad Nasr. Query-based adversarial prompt generation. 2024. URL https://arxiv.org/abs/2402.12329.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Liwei Jiang, Kavel Rao, Seungju Han, Allyson Ettinger, Faeze Brahman, Sachin Kumar, Niloofar Mireshghallah, Ximing Lu, Maarten Sap, Yejin Choi, and Nouha Dziri. Wildteaming at scale: From in-the-wild jailbreaks to (adversarially) safer language models. In *Advances in Neural Information Processing Systems*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

Zeyi Liao and Huan Sun. Amplegcg: Learning a universal and transferable generative model of adversarial suffixes for jailbreaking both open and closed llms. *arXiv preprint arXiv:2404.07921*, 2024.

Mantas Mazeika, Long Phan, Xuwang Yin, Andy Zou, Zifan Wang, Norman Mu, Elham Sakhaee, Nathaniel Li, Steven Basart, Bo Li, David Forsyth, and Dan Hendrycks. HarmBench: A standardized evaluation framework for automated red teaming and robust refusal, 2024. URL https://arxiv.org/abs/2402.04249.

Anselm Paulus, Arman Zharmagambetov, Chuan Guo, Brandon Amos, and Yuandong Tian. Advprompter: Fast adaptive adversarial prompting for llms. 2025. URL https://arxiv.org/abs/2404.16873.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Vinu Sankar Sadasivan, Shoumik Saha, Gaurang Sriramanan, Priyatham Kattakinda, Atoosa Chegini, and Soheil Feizi. Fast adversarial attacks on language models in one gpu minute. 2024. URL https://arxiv.org/abs/2402.15570.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. URL https://arxiv.org/abs/2307.09288.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition, 2024. URL https://arxiv.org/abs/2402.15220.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Sicheng Zhu, Ruiyi Zhang, Bang An, Gang Wu, Joe Barrow, Zichao Wang, Furong Huang, Ani Nenkova, and Tong Sun. Autodan: Interpretable gradient-based adversarial attacks on large language models, 2023. URL https://arxiv.org/abs/2310.15140.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

A Implementation Details of Jailbreak Attacks

Attack Methods. Our experiments adopt six token-level jailbreak attacks, including four optimization-based methods: GCG (Zou et al., 2023), BEAST (Sadasivan et al., 2024), GCQ (Hayase et al., 2024), and Zhu's AutoDAN (Zhu et al., 2023), as well as two model-based attacks: AmpleGCG (Liao & Sun, 2024) and AdvPrompter (Paulus et al., 2025). We re-implemented all six evaluated attacks to apply PSKV and enable efficient batched operations. Unless otherwise stated, the jailbreak suffix length is 20 tokens; Appendix E.3 also evaluates longer suffixes. The hyperparameters for each attack are configured as follows:

- **GCG:** Following Algorithm 1 in Zou et al. (2023) and its official repository, we set: iterations = 500, top-k = 256, batch size (search width) = 64, prompt batch size = 16, and search width batch size = 16.
- **GCQ:** Following Algorithm 1 in Hayase et al. (2024), we set: iterations = 500, top-k = 256, proxy batch size and query batch size = 15, buffer size (search width) = 64, and batch size for prompts and buffer = 16.
- **BEAST:** Following Algorithm 1 in Sadasivan et al. (2024) and its official repository, we set: both beam-search parameters = 15, and beam-search batch size = 15.
- **Zhu's AutoDAN:** Following Algorithms 1 and 2 in Zhu et al. (2023), we set: iterations = 3, objective weights = (10, 100), top-B = 256, and temperature = 2.
- **AmpleGCG:** Following Algorithm 1 in Liao & Sun (2024) and its official repository, we set: steps = 500, batch size (search width) = 64, and prompt batch size = 16.
- **AdvPrompter:** Following Algorithm 1 in Paulus et al. (2025) and its official repository, we set: epochs = 10, candidate set size = 256, prompt batch size = 8, temperature = 0.6, penalty parameter = 1.2, beam set size = 64, and beam number = 4.

Attack Success Rate(ASR). For each sample from the safety dataset, we perform a jailbreak attack to synthesize a jailbreak suffix and use it to induce the targeted LLMs to generate 10 responses. An LLM-based judger [4] from Mazeika et al. (2024) is then used to evaluate whether a response contains harmful information. An attack is determined to be successful if any of the 10 responses is determined to be harmful.

B Proof: time and memory complexity

B.1 Complexity Analysis for Single Prompt

In this section, we provide the derivations for the computational (C) and memory (M) complexities presented in Section 4.2. We let E be the number of synthesis iterations; N_p, N_s, N_t be the token lengths of the prefix, suffix, and target, respectively; and $K \cdot q$ be the number of candidates evaluated per iteration. Our analysis abstracts away model-specific constants (e.g., hidden dimension, number of layers) as they are common factors across all methods. The core assumptions are that the attention mechanism has a computational cost of $\mathcal{O}(L^2)$ for a sequence of length L, and the corresponding KV cache has a memory footprint of $\mathcal{O}(L)$.

B.1.1 Method without Acceleration

Computational Complexity. This method recomputes the entire sequence of length $N_p + N_s + N_t$ for each of the $K \cdot q$ candidates in every one of the E iterations. The total complexity is the product of the number of operations and the cost per operation:

$$\begin{aligned} C_{\text{NoAccel}} &= E \cdot (K \cdot q) \cdot \mathcal{O}((N_p + N_s + N_t)^2) \\ &= \mathcal{O}(E \cdot (K \cdot q) \cdot (N_p + N_s + N_t)^2). \end{aligned} \tag{4}$$

[4] https://huggingface.co/cais/HarmBench-Llama-2-13b-cls

Memory Complexity. The peak memory is determined by the batched inference step, which must hold the KV caches for all $K \cdot q$ candidates simultaneously. The memory for each cache is proportional to the full sequence length.

$$\begin{aligned} M_{\text{NoAccel}} &= (K \cdot q) \cdot \mathcal{O}(N_p + N_s + N_t) \\ &= \mathcal{O}((K \cdot q) \cdot (N_p + N_s + N_t)). \end{aligned} \tag{5}$$

B.1.2 Standard KV Cache

Computational Complexity. This method involves a one-time prefix computation and an iterative evaluation step over E iterations.

- **Prefix Computation (One-time):** A single forward pass over the prefix of length N_p costs $\mathcal{O}(N_p^2)$.
- **Iterative Evaluation (per epoch):** For each of the $N_s + N_t$ tokens of the suffix and target, attention is computed over a context that grows from N_p to $N_p + N_s + N_t - 1$. The average context length for these tokens is approximately $N_p + (N_s + N_t)/2$. The computational cost for each of the $K \cdot q$ candidates is thus roughly proportional to the number of new tokens times twice the average context length, giving $\mathcal{O}((N_s + N_t) \cdot (2N_p + N_s + N_t))$.

The total complexity is the sum of the one-time cost and the total iterative cost:

$$\begin{aligned} C_{\text{Std}} &= \mathcal{O}(N_p^2) + E \cdot (K \cdot q) \cdot \mathcal{O}((N_s + N_t)(2N_p + N_s + N_t)) \\ &= \mathcal{O}(N_p^2 + E \cdot (K \cdot q) \cdot (N_s + N_t)(2N_p + N_s + N_t)). \end{aligned} \tag{6}$$

Memory Complexity. To achieve parallel computation, the pre-computed prefix cache of length N_p must be physically duplicated $K \cdot q$ times. The total memory is the sum of these duplicated prefix caches and the caches for the suffix and target segments.

$$\begin{aligned} M_{\text{Std}} &= (K \cdot q) \cdot \mathcal{O}(N_p) + (K \cdot q) \cdot \mathcal{O}(N_s + N_t) \\ &= \mathcal{O}((K \cdot q) \cdot (N_p + N_s + N_t)). \end{aligned} \tag{7}$$

B.1.3 PSKV (Ours)

Computational Complexity. The computational process of PSKV is identical to the standard KV cache approach, as it also performs a one-time prefix computation followed by iterative evaluation of the suffix and target segments.

$$C_{\text{Ours}} = C_{\text{Std}} = \mathcal{O}(N_p^2 + E \cdot (K \cdot q) \cdot (N_s + N_t)(2N_p + N_s + N_t)). \tag{8}$$

Memory Complexity. The key advantage of PSKV is that it avoids the physical duplication of the prefix cache. It stores the prefix cache of length N_p only once, alongside the caches for the $K \cdot q$ suffix and target segments.

$$\begin{aligned} M_{\text{Ours}} &= \mathcal{O}(N_p) + (K \cdot q) \cdot \mathcal{O}(N_s + N_t) \\ &= \mathcal{O}(N_p + (K \cdot q) \cdot (N_s + N_t)). \end{aligned} \tag{9}$$

B.2 Derivations for Batched Complexity Analysis

In this section, we provide the derivations for the complexities of batched suffix synthesis presented in Section 4.3. We let B be the number of prompts, E be the number of synthesis iterations, and $K \cdot q$ be the candidates per prompt per iteration. The total candidates per iteration is $B \cdot K \cdot q$. Let $N_{p,\text{max}}, N_s, N_{t,\text{max}}$ be the maximum token lengths of the aligned prefix, suffix, and target, and $N_{p,\text{avg}}, N_{t,\text{avg}}$ be the average lengths.

B.2.1 Method without Acceleration

Computational Complexity. This method recomputes the full sequence for every candidate in every iteration. The total number of sequences processed is $E \cdot B \cdot K \cdot q$. The cost for each is quadratic to its length.

$$\begin{aligned} C_{\text{NoAccel}} &= E \cdot (B \cdot K \cdot q) \cdot \mathcal{O}((N_{p,\text{avg}} + N_s + N_{t,\text{avg}})^2) \\ &= \mathcal{O}(E \cdot (B \cdot K \cdot q) \cdot (N_{p,\text{avg}} + N_s + N_{t,\text{avg}})^2). \end{aligned} \tag{10}$$

Memory Complexity. The peak memory is determined by the batched inference step, which holds the KV caches for all $B \cdot K \cdot q$ candidates simultaneously.

$$\begin{aligned} M_{\text{NoAccel}} &= (B \cdot K \cdot q) \cdot \mathcal{O}(N_{p,\text{avg}} + N_s + N_{t,\text{avg}}) \\ &= \mathcal{O}((B \cdot K \cdot q) \cdot (N_{p,\text{avg}} + N_s + N_{t,\text{avg}})). \end{aligned} \tag{11}$$

B.2.2 Standard KV Cache

Computational Complexity. This method pre-computes the cache for the B prefixes once, then iteratively evaluates the suffixes and targets.

- **Prefix Computation (One-time):** A batched pass over B prefixes of max length $N_{p,\text{max}}$ costs $\mathcal{O}(B \cdot N_{p,\text{max}}^2)$.
- **Iterative Evaluation (per epoch):** For each of the $B \cdot K \cdot q$ candidates, processing the $N_s + N_{t,\text{max}}$ new tokens costs approximately $\mathcal{O}((N_s + N_{t,\text{max}}) \cdot (2N_{p,\text{max}} + N_s + N_{t,\text{max}}))$.

The total complexity over E iterations is the sum of the one-time cost and the total iterative cost:

$$C_{\text{Std}} = \mathcal{O}(B \cdot N_{p,\text{max}}^2 + E \cdot (B \cdot K \cdot q) \cdot (N_s + N_{t,\text{max}})(2N_{p,\text{max}} + N_s + N_{t,\text{max}})). \tag{12}$$

Memory Complexity. To process in a single batch, the B prefix caches must be duplicated $K \cdot q$ times each. The total memory is the sum of these duplicated prefix caches and the caches for the suffix/target segments.

$$\begin{aligned} M_{\text{Std}} &= (B \cdot K \cdot q) \cdot \mathcal{O}(N_{p,\text{max}}) + (B \cdot K \cdot q) \cdot \mathcal{O}(N_s + N_{t,\text{max}}) \\ &= \mathcal{O}((B \cdot K \cdot q) \cdot (N_{p,\text{max}} + N_s + N_{t,\text{max}})). \end{aligned} \tag{13}$$

B.2.3 PSKV (Ours)

Computational Complexity. The computational process is identical to the standard KV cache approach.

$$C_{\text{Ours}} = C_{\text{Std}} = \mathcal{O}(B \cdot N_{p,\text{max}}^2 + E \cdot (B \cdot K \cdot q) \cdot (N_s + N_{t,\text{max}})(2N_{p,\text{max}} + N_s + N_{t,\text{max}})). \tag{14}$$

Memory Complexity. PSKV avoids physical duplication. It stores one cache for each of the B prefixes, and the caches for the $B \cdot K \cdot q$ suffix/target segments.

$$\begin{aligned} M_{\text{Ours}} &= B \cdot \mathcal{O}(N_{p,\text{max}}) + (B \cdot K \cdot q) \cdot \mathcal{O}(N_s + N_{t,\text{max}}) \\ &= \mathcal{O}(B \cdot N_{p,\text{max}} + (B \cdot K \cdot q) \cdot (N_s + N_{t,\text{max}})). \end{aligned} \tag{15}$$

B.3 Dependence on sequence lengths, search width, and model size

The preceding complexity expressions isolate attention computation and cache storage. Here we distinguish these terms from total runtime and peak device memory. Let $p = N_p$, $s = N_s$, $t = N_t$, $r = s + t$, and $W = Kq$. Let B be the number of prompts whose prefix caches are retained, $b \le B$ the prompt microbatch size, and $w \le W$ the candidate microbatch width. Model dimensions are the number of layers N, hidden size H, vocabulary size V, and aggregate per-layer KV-head dimension $H_{\text{KV}} = n_{\text{KV}} d_{\text{head}}$. Let a denote bytes per stored scalar; BF16 has $a = 2$. For aligned dense batches, sequence lengths below refer to padded lengths.

Prefix, suffix, and target lengths. Under the attention-cost model of Appendix B.1, uncached candidate evaluation scales as $EBW(p+r)^2$, whereas both cached methods scale as $Bp^2 + EBWr(2p+r)$. The ratio of these cost expressions is

$$R_{\text{attn}} = \frac{(p+r)^2}{p^2/(EW) + r(2p+r)}. \tag{16}$$

This is an attention-work estimate with common constants suppressed, not a wall-clock speedup prediction. For many candidate evaluations, the one-time prefill is amortized. A larger prefix then increases the fraction of work that can be reused, while longer suffixes or targets increase unavoidable

candidate-dependent work and reduce the relative gain at a fixed prefix length. Prefix caching also avoids repeated token projections and feed-forward computation on the prefix; these costs, memory traffic, and attack-side operations are omitted from Eq. (16).

For an end-to-end interpretation, let f be the fraction of uncached runtime spent in the operations being accelerated, ρ their local speedup, and δ additional overhead normalized by uncached runtime. Then

$$S_{\text{total}} = \frac{1}{(1-f) + f/\rho + \delta}. \tag{17}$$

For nonnegative overhead, even arbitrarily fast execution of this component cannot exceed $1/(1-f)$ when $f < 1$. PSKV and Standard KV have the same leading arithmetic cost; PSKV primarily removes persistent candidate-wise cache replication. Its principal advantage over Standard KV is therefore memory and the ability to execute wider searches that otherwise do not fit.

Resident cache and transient workspace. Counting both keys and values, eager Standard KV and PSKV retain respectively

$$M_{\text{prefix}}^{\text{Standard}} = 2aBWNpH_{\text{KV}}, \tag{18}$$

$$M_{\text{prefix}}^{\text{PSKV}} = 2aBNpH_{\text{KV}}. \tag{19}$$

The resident prefix saving is $2aBNpH_{\text{KV}}(W-1)$ bytes. Thus it grows linearly with prefix length, prompt count, layer count, and KV-head dimension. Grouped-query attention decreases H_{KV} and consequently the absolute bytes saved. PSKV removes the dependence on candidate width from the resident prefix term, but it does not remove dependence on the number or length of distinct prompts.

In forward-only candidate scoring, expanding and concatenating the current layer's KV tensors requires workspace of order

$$M_{\text{workspace}} = \mathcal{O}(abw(p+r)H_{\text{KV}}), \tag{20}$$

in addition to activations, logits, and other working tensors. The expansion operation may initially produce a view, but reshaping or concatenating tensors can materialize storage. Consequently, the compact resident-cache expression should not be interpreted as a bound on total peak memory independent of p or w. Micro-batching bounds this workspace while retaining all B compact prefix caches. It also explains why increasing a dataset's size can increase resident cache memory without proportionally increasing the activation peak.

Suffix gradients and model scaling. In our GCG implementation, target-model parameters and prefix/target embeddings are frozen, and gradients are requested only for the suffix one-hot tensor. Model-parameter gradients are therefore not allocated. For a gradient microbatch of b prompts, storing this tensor and its gradient in BF16 takes

$$M_{\text{suffix}} = 4bsV \quad \text{bytes}. \tag{21}$$

For $b = 8$ and $s = 20$, this is approximately 0.091 GiB for Qwen2.5 ($V = 151{,}936$) and 0.076 GiB for Llama-3 ($V = 128{,}256$). These values concern only the one-hot tensor and its returned gradient, not the full backward graph or temporary one-hot construction buffers.

The suffix/target hidden-state and logit contributions to the backward graph scale approximately as

$$M_{\text{hidden+logits}} = \mathcal{O}(abN(s+t)H) + \mathcal{O}(ab(s+2t)V), \tag{22}$$

with implementation-dependent constants. FlashAttention (Dao et al., 2022) avoids materializing the full attention-score matrix, but backward computation still needs attention operands. In particular, saved concatenated KV tensors can introduce an additional $\mathcal{O}(abN(p+s+t)H_{\text{KV}})$ term; the one-layer workspace lifetime of forward-only scoring does not automatically apply to tensors retained for backward. These terms must be distinguished from parameter-gradient storage.

BF16 model weights occupy approximately $2P$ bytes for P parameters. Within an architectural family with fixed head ratios and lengths, transformer-block weights grow approximately as NH^2, whereas hidden-state and KV terms grow approximately linearly with H. This comparison explains why larger parameter count alone does not imply that parameter-sized gradients erase PSKV's benefit. It does not guarantee a monotonic improvement in the fraction of total memory saved: vocabulary size, sequence lengths, attention architecture, and the peak-producing phase all matter. This argument concerns the frozen target model in GCG; it does not apply to trainable generator parameters in AdvPrompter.

Search width and the observed peak. A useful phase-wise description is

$$M_{\text{peak}} \simeq M_{\text{weights}} + M_{\text{prefix}} + \max\{M_{\text{prefill}}, M_{\text{gradient}}, M_{\text{scoring}}\} + M_{\text{other}}, \tag{23}$$

where phase terms denote working memory beyond the weights and resident prefix cache, and M_{other} includes attack buffers and allocator overhead. GCG differentiates the current suffix before scoring its candidates, so its gradient microbatch does not grow with W. If that phase dominates and the scoring microbatch width stays bounded, the peak may remain almost unchanged as search width increases, as in Table 6. This is a property of the measured regime rather than a general claim that candidate search uses constant memory; candidate buffers and scoring work can eventually dominate.

C Computational Efficiency Analysis of AdvPrompter

C.1 Observations

In our experimental evaluation of AdvPrompter (Table 4), we observed two notable phenomena:

1. **Severe performance degradation without KV caching:** All models exhibit substantially higher latency in the "None" (no acceleration) setting compared to cached configurations. For instance, Llama-2-7B increases from 275s (Standard KV Cache) to over 500s (None), representing an approximately $2\times$ slowdown.
2. **Model-specific latency disparities:** Among models of comparable size, Llama-2-7B and Llama-3-8B consistently exhibit higher execution times than Mistral-7B and Qwen2.5-7B across all acceleration settings. Even with our PSKV, Llama-2 (250s) and Llama-3 (282s) remain slower than Mistral (80s) and Qwen (76s).

This section analyzes the algorithmic and architectural factors underlying these observations.

C.2 The Cost of Cache-less Decoding

The severe slowdown in the "None" setting stems from the fundamental inefficiency of autoregressive generation without KV caching.

Quadratic Total Complexity. With KV caching, generating token t requires computing attention only between the new query and cached keys/values, yielding $O(1)$ complexity per layer with respect to history length. Without caching, the model must recompute all previous hidden states, requiring $O(t)$ operations at step t. Summing over a sequence of length n, the total complexity degrades from $O(n)$ to $O(n^2)$.

Amplification by AdvPrompter's Search Strategy. Unlike gradient-based attacks (*e.g.*, GCG) that perform single forward-backward passes, AdvPrompter employs beam search with sampling during suffix generation. This expands the effective batch size multiplicatively (batch $\times$ beam $\times$ samples), causing the quadratic recomputation penalty to be incurred across many parallel sequences simultaneously. The resulting computational burden saturates GPU throughput and explains the disproportionate slowdown observed in the "None" setting.

C.3 Architectural Factors in Llama-Series Latency

While all models benefit from KV caching, Llama-2 and Llama-3 exhibit consistently higher latency than Mistral and Qwen. We attribute this to two architecture-specific factors.

Llama-2: Multi-Head Attention Overhead. Llama-2-7B employs Multi-Head Attention (MHA) with 32 key-value heads, whereas Mistral-7B and Qwen2.5-7B utilize Grouped-Query Attention (GQA) with 8 key-value heads. In cache-less decoding, the model must recompute K and V projections for the entire sequence history at each generation step. With $4\times$ more KV heads, Llama-2 incurs proportionally higher memory bandwidth requirements for loading projection weights and storing intermediate KV tensors. This memory-bound overhead is particularly pronounced when batch sizes are large, as in AdvPrompter's beam search.

Llama-3: Vocabulary Projection Overhead. Llama-3-8B adopts GQA, mitigating the MHA-related overhead. However, it employs a vocabulary of 128K tokens—approximately $4\times$ larger than Llama-2 and Mistral ($\sim$32K). AdvPrompter uniquely requires training a generator model, which involves computing full logit distributions over the vocabulary and backpropagating through the output projection layer. The computational cost of this $\mathbb{R}^d \to \mathbb{R}^{|V|}$ projection and its gradient scales linearly with vocabulary size, imposing a persistent overhead on Llama-3 that cannot be alleviated by KV caching alone.

The "None" baseline forces redundant recomputation across the entire sequence history at each decoding step, a cost amplified by AdvPrompter's beam search strategy. Llama-2 is further penalized by its MHA architecture, which demands higher memory bandwidth than GQA-based models. Llama-3, while adopting GQA, suffers from the computational burden of its large vocabulary during AdvPrompter's generator training phase. Our PSKV effectively eliminates redundant prefix computations, providing consistent speedups across all evaluated models.

D Detailed analysis of memory overhead

To substantiate the memory efficiency analysis presented in the main text, we provide a detailed breakdown of the model architectures, the mathematical formulation of KV cache memory footprint, and the specific implementation differences between Standard KV Cache and PSKV.

D.1 Architectural Specifications of Target Models

The effectiveness of PSKV is intrinsically linked to the underlying attention mechanism and vocabulary size of the target Large Language Models (LLMs). Table 8 summarizes the critical architectural parameters for the models evaluated in this study.

Table 8: Architectural specifications of the evaluated models. **MHA**: Multi-Head Attention; **GQA**: Grouped-Query Attention; **SWA**: Sliding Window Attention.

Model	Attention Type	Vocab Size	Q Heads	KV Heads	KV Compression
Vicuna-1.5-7B	MHA	32,000	32	32	1:1
Llama-2-7B	MHA	32,000	32	32	1:1
Llama-3-8B	GQA	128,256	32	8	4:1
Mistral-7B-v0.1	GQA + SWA	32,000	32	8	4:1
Qwen2.5-7B	GQA	151,936	28	4	7:1

Impact of Attention Mechanisms (MHA vs. GQA). The memory footprint of the KV cache is determined by the number of Key-Value heads.

- **Multi-Head Attention (MHA):** Used by Llama-2 and Vicuna. Here, $H_{kv} = H_q$. Every query head requires a unique key-value pair, leading to maximum memory usage.
- **Grouped-Query Attention (GQA):** Used by Llama-3, Mistral, and Qwen. Here, $H_{kv} < H_q$. Multiple query heads share a single KV head (e.g., Qwen utilizes 28 query heads but only 4 KV heads). This architectural change physically reduces the KV cache tensor size by a factor of H_q/H_{kv}, which inherently lowers the memory ceiling and thus reduces the *relative* impact of PSKV compared to MHA models.

Impact of Vocabulary Size. The memory consumption during attack generation is not solely due to the KV cache but also intermediate activation tensors, specifically the logits tensor, which has shape $B \times T \times |\mathcal{V}|$. As shown in Table 8, Llama-3 and Qwen2.5 have vocabulary sizes $|\mathcal{V}|$ approximately $4\times$ to $5\times$ larger than Llama-2. This drastically increases the non-KV memory overhead (e.g., storing logits and gradients), further diluting the proportion of memory saved by PSKV.

Note on Mistral's Sliding Window. Mistral-7B employs Sliding Window Attention (SWA) with a large window size. While SWA theoretically caps the KV cache size to the window length, our

experiments use suffix attack sequences whose total length is smaller than this window. Therefore, the cache behavior in our experiments effectively mirrors standard GQA without the sliding window truncation benefits.

D.2 Theoretical Memory Complexity: Standard vs. PSKV

We formalize the memory consumption to explain why Standard KV Cache results in OOM errors under beam search settings while PSKV remains efficient. Let B be the batch size (or beam width/number of candidates), L_{pre} be the prefix length, L_{gen} be the generated-token length, D be the per-head dimension, H_{kv} be the number of KV heads, and N_{layer} be the number of layers.

Standard KV Cache (Eager Expansion). Standard implementations (e.g., Hugging Face default) eagerly expand the KV cache to match the batch dimension at initialization. The memory complexity for the cached prefix is:

$$M_{std} \propto N_{layer} \times \mathbf{B} \times L_{pre} \times H_{kv} \times D \tag{24}$$

Under optimization-based attacks (e.g., GCQ, BEAST), B represents the number of candidates. The $B \times L_{pre}$ term causes memory usage to scale linearly with the candidate count, leading to rapid exhaustion of GPU memory.

PSKV (Layer-wise Dynamic Expansion). Our PSKV implementation maintains a singleton copy of the prefix cache. During forward-only candidate scoring at the ℓ-th layer, the cache is virtually broadcasted or temporarily expanded for that layer's computation; the expansion need not persist across layers. The resident prefix-cache footprint is:

$$M_{pskv} \propto N_{layer} \times \mathbf{1} \times L_{pre} \times H_{kv} \times D \tag{25}$$

The temporary expansion memory required during the forward pass is $\mathcal{O}(B \cdot L_{pre} \cdot H_{kv} \cdot D)$ for the current layer computation. This allows the resident prefix-cache storage to remain independent of candidate count, providing the efficiency gains observed in the main results. Appendix B.3 distinguishes this resident storage from forward workspace and tensors retained for backward.

D.3 Attack-Specific Memory Overhead Details

Different attack algorithms impose different auxiliary memory costs that interact with the cache mechanism:

- **High-Width Search (GCQ, BEAST):** These algorithms generate a large number of candidate sequences per step. While PSKV compresses the prefix storage, the intermediate activations (e.g., attention scores, FFN intermediate states) still scale with B. Thus, while PSKV prevents OOM from the *cache* perspective, the *activation* memory creates a new floor for memory usage.
- **Dual-Model Loading (AdvPrompter):** AdvPrompter requires the simultaneous residence of the Target LLM and the Attacker LLM (Prompter) in VRAM. This effectively halves the available memory for dynamic tensors. In this constrained environment, the static weight of the models dominates the memory profile, making the dynamic savings from PSKV less visually prominent in percentage terms compared to single-model attacks.

E Additional Experiments and Analyses

E.1 ASR stability and numerical equivalence

PSKV is intended to be an efficiency-only change: it shares prefix KV tensors rather than changing the attack objective, candidate generation rule, or decoding/evaluation policy. To test whether the ASR fluctuations in the original single-run table were caused by PSKV, we ran additional paired multi-seed checks on the attack/model combinations with the largest observed fluctuations.

All observed differences fall within one standard deviation. A paired ten-seed GCG check further produced $t = 0.35$, $p = 0.74$, a 95% confidence interval of $[-0.022, +0.030]$, and Cohen's $d = 0.12$,

Table 9: Additional multi-seed ASR checks for high-variance attack/model pairs. Values are mean ± standard deviation over five seeds, complementing the three-run results in Table 3.

Attack–Model	No Cache	PSKV
GCG–Llama-2	66.0 ± 3.44	64.4 ± 3.35
GCQ–Vicuna	91.6 ± 3.20	92.0 ± 4.20
BEAST–Llama-2	38.0 ± 3.58	38.0 ± 3.53
BEAST–Llama-3	41.2 ± 4.83	40.0 ± 4.38
AdvPrompter–Llama-3	76.4 ± 4.27	77.2 ± 3.71

indicating a negligible effect size. We also compared logits across No Cache, Standard KV Cache, and PSKV on identical inputs. In FP32/FP64, all modes produced bit-identical logits. In BF16, cached and uncached modes can differ because of reduced precision, but Standard KV Cache and PSKV produced identical logits. This supports the claim that ASR variation is due to the attacks' stochastic components rather than an approximation introduced by PSKV.

E.2 Larger models and evaluation sets

We additionally evaluate GCG on Llama-2-13B with 50 HarmBench examples and on Llama-2-7B with 50, 100, and all 200 examples of the HarmBench evaluation set. Within each comparison, None, Standard KV, and PSKV use identical attack settings and hardware. All prompts belong to one batched attack run; gradient computation and candidate scoring are microbatched, while one prefix cache per prompt is retained under PSKV. These controlled runs are reported separately from the main benchmark, and comparisons below are made within each setting.

Table 10: GCG with a larger target model and larger HarmBench evaluation sets. Time is total attack runtime in minutes; memory is peak memory in GB. OOM denotes an out-of-memory failure.

Model	Cache	Examples	Time (min)	Memory (GB)
Llama-2-13B	None	50	164	42.7
	Standard KV	50	OOM	OOM
	PSKV	50	108	38.2
Llama-2-7B	None	50	97	23.9
	Standard KV	50	65	98.4
	PSKV	50	65	23.1
Llama-2-7B	None	100	187	24.5
	Standard KV	100	125	145.0
	PSKV	100	126	24.5
Llama-2-7B	None	200	387	25.9
	Standard KV	200	OOM	OOM
	PSKV	200	262	26.8

On Llama-2-13B, PSKV reduces runtime from 164 to 108 minutes ($1.52\times$ speedup) and peak memory from 42.7 to 38.2 GB (10.5%) relative to None, while Standard KV runs out of memory. Across the 50-, 100-, and 200-example Llama-2-7B settings, the speedup over None remains 1.48–$1.49\times$. Where Standard KV fits, PSKV has nearly identical runtime and reduces its peak memory by 76.5% and 83.1%, respectively. At 200 examples, PSKV completes the evaluation with 26.8 GB while Standard KV fails. PSKV's peak is slightly higher than None in this last setting (26.8 versus 25.9 GB), consistent with retaining one cache per prompt; prefix sharing does not guarantee a lower total peak than uncached execution in every configuration.

E.3 Longer suffixes and longer prompts

We also evaluated GCG on Llama-2-7B with longer suffix lengths and with longer-prefix inputs from WildJailbreak (Jiang et al., 2024). Table 11 reports peak memory and runtime. Longer suffixes increase the cost for all methods because suffix/target computation remains unavoidable. Longer

prefixes, however, make PSKV more favorable because it stores one prefix cache and expands it layer by layer, while Standard KV Cache stores a fully expanded prefix cache.

Table 11: GCG scaling with longer suffixes and prompts on Llama-2-7B. Entries are memory in GB / time in minutes.

Setting	No Cache	Standard KV	PSKV
SFX40	113 / 255	140 / 190	80 / 190
SFX60	116 / 306	147 / 230	94 / 230
SFX20–WildJailbreak	126 / 363	OOM	64 / 206

E.4 Universal suffix optimization

For a universal suffix, the same suffix is optimized across multiple prompt–target pairs. Writing $\ell_i(s) = -\log p_\theta(y_i^{(h)} \mid x_i^{(h)} \oplus s)$, the objective becomes

$$s^* = \arg\min_{s \in \mathcal{V}^{N_s}} \frac{1}{B} \sum_{i=1}^{B} \ell_i(s). \tag{26}$$

Each prompt retains its own fixed prefix, so PSKV stores one prefix cache per prompt and reuses it across that prompt's candidate evaluations. Suffix-centric alignment accommodates different prompt lengths, while the shared suffix and the aggregation of losses or gradients across prompts remain unchanged. For optimization involving multiple models, caches must be constructed separately for each model because their parameters and internal representations differ; PSKV can be applied within each model independently.

We perform a focused Universal GCG experiment on Llama-2-7B: one shared suffix is optimized on 10 HarmBench prompts and subsequently evaluated on 40 held-out prompts. None, Standard KV, and PSKV use identical initialization and search settings. Table 12 reports search cost; it does not report held-out ASR or establish cross-model transfer effectiveness.

Table 12: Search cost of Universal GCG on Llama-2-7B with 10 optimization prompts.

Cache	Search time (min)	Peak memory (GB)
None	16.92	24.14
Standard KV	12.73	33.29
PSKV	12.41	21.44

PSKV provides a $1.36\times$ speedup over None (26.7% less search time) and reduces peak memory by 11.2%. Relative to Standard KV, runtime is similar while peak memory decreases by 11.85 GB (35.6%). This experiment extends the observed compute–memory benefit to shared-suffix optimization across multiple prompts.

E.5 Relationship to serving engines and Faster-GCG

Serving engines such as vLLM and SGLang solve a different systems problem. They are optimized for asynchronous, multi-tenant forward inference with request scheduling, page management, prefix lookup, and eviction policies. Their custom attention kernels do not provide the native backward path required by white-box suffix attacks such as GCG, GCQ, AutoDAN, AmpleGCG, and AdvPrompter. PSKV instead stays inside PyTorch's autograd graph and specializes the one-prefix-many-suffixes workload by expanding the shared prefix cache lazily at each layer.

Faster-GCG is also orthogonal to PSKV. Faster-GCG changes the search algorithm for GCG-style attacks, whereas PSKV accelerates the model-side forward computations used by suffix attacks. In principle, an implementation could combine Faster-GCG's search procedure with PSKV's prefix sharing.

E.6 PLUG-AND-PLAY INTEGRATION

PSKV can be integrated into an existing PyTorch attack loop with a small number of local changes: initialize the prefix cache once, replace the ordinary model forward call for candidate scoring with the PSKV-aware forward function, and pass the candidate expansion factor used by the scoring loop. The larger re-implementations used in our experiments were for a unified benchmark, not because PSKV requires rewriting each attack from scratch.

E.7 GRADIENT CHECKPOINTING AND MICRO-BATCHING

Gradient checkpointing and micro-batching are complementary to PSKV. Under No Cache, the gradient forward pass can be dominated by prefix-related intermediate activations, so checkpointing can reduce the peak. Under PSKV, layer-wise lazy expansion already removes much of that prefix footprint; at longer suffix lengths or larger candidate counts, scoring tensors and candidate suffix activations can dominate instead, limiting what checkpointing can save. Micro-batching remains useful for both scoring and gradient computation because it reduces candidate-level activation peaks. These techniques can be combined with PSKV when the workload is limited by non-cache activation memory.

E.8 RESPONSIBLE RELEASE

PSKV is a dual-use optimization because it lowers the compute cost of running existing jailbreak attacks. The intended use is scalable safety auditing. A responsible release should provide the acceleration library and reproducibility instructions while avoiding bundled harmful prompt datasets or turnkey malicious attack scripts. Access can be gated by a research-only license and acceptable-use policy, following the precedent of safety-evaluation resources such as HarmBench and GCG.