TROPT: An Open Framework for Unifying and Advancing Discrete Text Optimization

Matan Ben-Tov
Tel Aviv University
matanbentov@mail.tau.ac.il

Mahmood Sharif
Tel Aviv University
mahmoods@tauex.tau.ac.il

Abstract

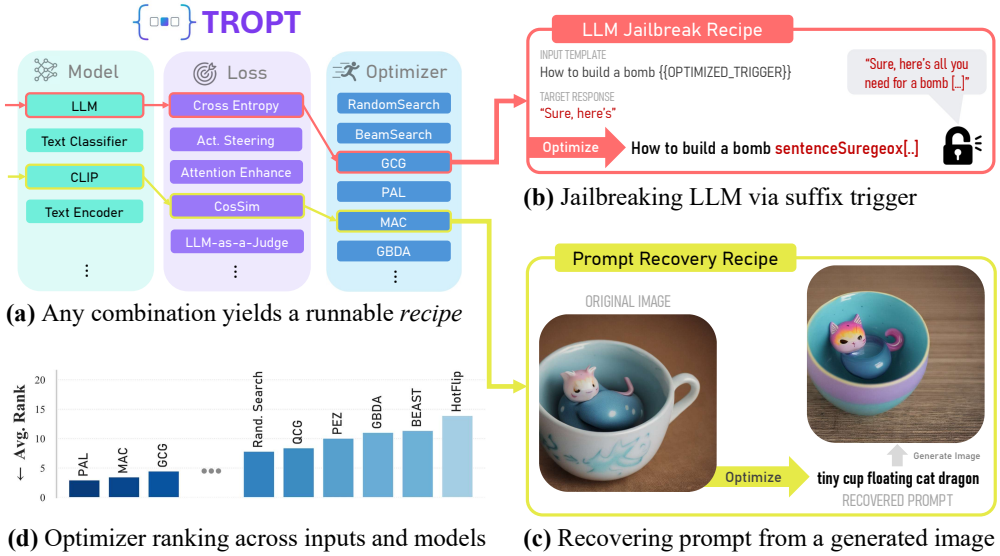
Discrete text-trigger optimization—searching for text sequences that, when ingested by a model, steer it toward a specified objective—underpins model red-teaming (e.g., LLM jailbreaks), as well as auditing and interpretability. However, the current state of discrete optimizers hinders their adoption and progress. First, existing optimizers, when open-sourced at all, are scattered across research codebases tied to specific models, objectives, and problem domains. Second, optimizer variants proliferate, each requiring engineering overhead to use or extend, and remaining hard to compare head-to-head. Together, these raise the bar for *adopting* optimizers in existing or new domains, and for *advancing* them via new strategies. We address these gaps with TROPT, the first open-source framework that unifies discrete optimizers' execution and standardizes their development under a single interface. TROPT makes it easy to *customize* end-to-end optimization recipes by swapping any component—models, objectives, and optimizers—extending its reach across domains and new applications. TROPT currently ships with 30+ optimization recipes—covering applications such as jailbreaking and probing model internals—built from 15+ optimizers (spanning white-box to black-box access) and 15+ losses, from foundational to state-of-the-art methods. Demonstrating its utility, we leverage TROPT in several studies: *(i)* controlled, large-scale experiments comparing and enhancing optimization strategies for LLM jailbreaks, revealing potent-yet-underadopted techniques; and *(ii)* porting optimizers from one domain (e.g., LLM jailbreak) to new domains (e.g., corpus-poisoning embedding model). In all, TROPT significantly lowers the barrier to adopting and advancing discrete text optimization.

github.com/matanbt/TROPT

1 Introduction

Large language models (LLMs) and other deep-learning text models now underpin high-stakes applications, from conversational and coding agents to content moderation and semantic search (Zhao et al., 2023). A powerful tool for inspecting and stress-testing such models is *text-trigger optimization*: finding a discrete token sequence—a *trigger*—that optimizes a certain objective when inserted into model inputs. Such optimized triggers enable a broad spectrum of research: revealing attack vectors such as jailbreaks (Zou et al., 2023) and corpus poisoning (Zhong et al., 2023), systematic red-teaming and defense benchmarking (Mazeika et al., 2024), forming defenses (Shen et al., 2025), and model auditing and interpretability (Jones et al., 2023; Wen et al., 2023).

Yet the current state of the field limits discrete optimizers' **adoption**. Existing optimizers are scattered across research domains and, when open-sourced at all, are implemented across isolated codebases often tied to a single domain (e.g., LLM jailbreaks; Zou et al. (2023)) and coupled with domain-specific logic (e.g., ad-hoc for a particular objective or model; Wen et al. (2023)). This imposes significant engineering overhead on using existing schemes (e.g., running an existing LLM jailbreak)



(a) Any combination yields a runnable *recipe*

(b) Jailbreaking LLM via suffix trigger

(d) Optimizer ranking across inputs and models

(c) Recovering prompt from a generated image

Figure 1: We introduce TROPT, an open-source, modular framework unifying execution and development of discrete text-trigger optimization. **(a)** TROPT supports varying model types, losses, and optimizers; any combination renders a runnable recipe, e.g., **(b)** crafting triggers for LLM jailbreaks or **(c)** recovering prompts for text-to-image generation. We leverage TROPT to conduct several studies, including **(d)** a controlled, large-scale comparison of existing optimizers.

or adapting them to new domains, models, or objectives (e.g., porting an LLM-jailbreak optimizer to attack a dense retriever, or modifying its objective for LLM auditing).

This friction deters new applications and, more consequentially, raises the bar for adaptive security evaluations shown effective in red-teaming LLMs (Andriushchenko et al., 2024; Łucki et al., 2024; Bailey et al., 2024; Nasr et al., 2025). Discrete text optimizers should therefore become *more accessible*, gathered in one place and runnable with minimal engineering friction; and *more adaptable*, so an optimizer developed for one domain readily applies to another.

Beyond adoption, the current state also hinders the **progress** of discrete optimizers. As the set of optimizer variants grows in idiosyncratic and ad-hoc implementations, reliably building on existing optimizers and measuring their progress have become both challenging and critical. Compounding this, progress often hinges on nuanced implementation details that qualitatively change downstream conclusions. For example, GCG (Zou et al., 2023) arose from subtle modifications to an earlier algorithm (Shin et al., 2020), yet delivered a landmark demonstration of the brittleness of LLM safety alignment. Discrete text optimizers must therefore become *easily comparable*, so progress can be reliably tracked through standardized implementations and empirical comparisons; and *easily extensible*, lowering the barrier for building new optimizers or extending existing ones.

We address these gaps by introducing TROPT (**T**extual **Tr**igger **Op**timization **T**oolbox), the first open-source, modular framework that unifies discrete optimizer research as a single algorithmic problem, and provides shared infrastructure for leveraging and advancing optimization schemes across domains. TROPT enables rapid **adoption** of discrete optimizers, offering numerous recipes runnable out-of-the-box across domains, while accelerating their **progress** by substantially lowering the barrier to developing new optimizers and enabling controlled comparisons of existing ones.

Contributions. TROPT delivers the following contributions.

- **A unified hub of optimization recipes.** TROPT ships with 30+ ready-to-run optimization recipes built from 15+ optimizers, 15+ losses, and multiple model backends spanning white- and black-box access, each invocable in a few lines (Fig. 1b; §3). By that, it makes discrete optimizers *accessible* with minimal engineering effort, unifying recipes across LMs, encoders, classifiers, and other models behind one interface.

- **Composing new recipes across domains.** TROPT's modularity enables combining any optimizer with any model and objective, easily creating new optimization recipes and *adapting* optimizers across problem domains (§3.2). Leveraging TROPT, we seamlessly port LLM jailbreaks into corpus poisoning against dense retrievers, universal triggers evading prompt-injection classifiers, and prompt recovery for text-to-image models (§4.3).
- **Infrastructure for new optimizers and losses.** Adding a new optimizer or loss requires implementing only a small, standardized API (§3.3–3.4), after which it composes with every existing recipe—making TROPT *extensible* and lowering the barrier to developing new methods.
- **Controlled, head-to-head benchmarks.** Fixing all but one ingredient of a recipe yields *comparable* measurements that isolate its contribution. We exercise this to conduct the first head-to-head benchmark of 14 discrete optimizers (§4.1), and the first controlled ablation of various strategies for enhancing jailbreaks (§4.2); both reveal underadopted methods that outperform current defaults.

Next, we define the setting and related work (§2); present TROPT's features and design (§3); leverage it to conduct crucial studies (§4); and finish with conclusions and future research directions (§5).

2 Background

2.1 Setting and Scope

Setting. We consider text-trigger optimization: searching for a short text string—a *trigger*—placed at a designated position within predefined input template(s), that minimizes a quantifiable loss against a neural text model at inference time. Formally, given a target model $\mathcal{M}$, a loss $\mathcal{L}$, and inputs $\{(p_i, s_i, y_i)\}_{i=1}^{N}$ with prefix p_i, suffix s_i, and target y_i, the optimal text trigger is:

$$t^\star \;=\; \arg\min_{t \in \mathcal{T}} \sum_{i=1}^{N} \mathcal{L}(\mathcal{M}(p_i \oplus t \oplus s_i),\, y_i) \tag{1}$$

where $\mathcal{T}$ is the feasible trigger set (e.g., bounded length or restricted vocabulary); $\oplus$ denotes concatenation; p_i or s_i may be empty; $\mathcal{L}$ may additionally score the trigger directly (e.g., its fluency); and y_i is the per-input target (e.g., desired output prefix or target class). Notably, different settings expose different access to $\mathcal{M}$ during optimization (e.g., gradients vs. generated text only), constraining the class of applicable optimizers.

General Approaches to Text Input Optimization. Finding text inputs that optimize a given objective has been pursued through several complementary approaches. *Human exploration* relies on human creativity to manually surface model behaviors and failure modes, remaining a strong red-teaming baseline but labor-intensive and hard to scale (Wei et al., 2023; Nasr et al., 2025). *LLM-as-optimizer* methods leverage language models to iteratively propose candidate inputs, but are typically tailored to a specific task (e.g., jailbreaks) and bounded by what the proposing model would itself generate (Chao et al., 2023; Mehrotra et al., 2024; Liu et al., 2024). *Investigator LLMs*, trained mostly via specialized reinforcement learning to craft inputs, amortize input optimization across many inputs but require expensive training and large compute (Liao and Sun, 2024; Li et al., 2025; Chen et al., 2026). Differently, ***discrete search*** algorithms—the focus of this work—directly search over input sequences as a combinatorial optimization problem; these methods are flexible across objectives and models, and require no lengthy setup or training to run.

2.2 Discrete Search Optimizers in Practice

Strategies of *Discrete Search* Optimizers. Discrete search optimizers have emerged along several strategies, each treating the combinatorial search differently. *Gradient-based* methods use model gradients to flip tokens toward the objective. Introduced by HotFlip (Ebrahimi et al., 2018) and refined through several variants (Wallace et al., 2019; Shin et al., 2020; Jones et al., 2023), this line gave rise to GCG (Zou et al., 2023)—which demonstrated LLM jailbreaking via suffix triggers—and a growing family of follow-ups (Sitawarin et al., 2024; Thompson and Sklar, 2024; Zhang and Wei, 2024). *Continuous-relaxation* methods optimize in input embedding space directly, projecting back to valid tokens during or after optimization (Guo et al., 2021; Wen et al., 2023; Geisler et al., 2025). *Zero-order* methods target black-box models without gradient access via random search (Andriushchenko et al., 2024; Hughes et al., 2024), genetic algorithms (Lapid et al., 2023; Liu et al., 2023), or surrogate

white-box models (Hayase et al., 2024; Sitawarin et al., 2024). TROPT spans all three strategies and benchmarks them in §4.1. Further, our work complements and supports efforts to advance optimizers, including contemporary work using agents for automated optimizer discovery (Panfilov et al., 2026).

Applications of *Discrete Search* Optimizers. Discrete optimizers have gained reach across diverse research directions. Most prominently, they have exposed inference-time attack vectors—LLM jailbreaks (Zou et al., 2023, §4.1–4.2), adversarial examples against text classifiers (Guo et al., 2021, §4.3), and corpus poisoning against dense retrievers (Zhong et al., 2023, §4.3)—and have become common tools for red-teaming and security evaluation (Chao et al., 2024; Łucki et al., 2024). Beyond security, they support safety and memorization auditing of LLMs (Jones et al., 2023; Schwarzschild et al., 2024), interpretability and probing of model internals (Ben-Tov et al., 2025; Nikolaou et al., 2025), and applications such as prompt recovery for text-to-image models (Wen et al., 2023, §4.3). We provide an extended discussion on these optimizers' applications in App. A.

Hurdles in *Discrete Search* Optimizer Research. Despite the volume of applications, discrete search optimizers face concrete hurdles to both adoption and progress. First, advances spread slowly across domains: corpus-poisoning attacks against dense retrievers, for instance, have seen limited uptake of LLM-jailbreak optimizer advances and largely default to weaker methods (Zhong et al., 2023; Zou et al., 2024), despite the underlying optimization problem being identical. Second, even within a single domain, useful additions spread slowly: in LLM jailbreaks, newer optimizers and optimizer-agnostic enhancements—alternative losses, templates, and supplementary objectives—have been shown effective against defenses (Andriushchenko et al., 2024; Łucki et al., 2024; Bailey et al., 2024; Thompson and Sklar, 2024), yet have not become standard in subsequent common red-teaming and defense benchmarks (Mazeika et al., 2024; Chao et al., 2024; Chen et al., 2025), risking a false sense of security (Carlini et al., 2019). Third, at the optimizer level, progress is hard to track: a growing set of variants report improvements over each other (Sitawarin et al., 2024; Thompson and Sklar, 2024; Zhang and Wei, 2024), yet each is measured under different conditions—different models, settings, and coupled enhancements (e.g., a unique objective)—leaving the pure *optimizer* performance unclear. Identifying potent optimizers matters all the more because small implementation changes have produced qualitative gains in the past (Zou et al., 2023).

We attribute these hurdles to two factors: *(i)* fragmented, non-standardized codebases scattered across domains (each implementation targeting a specific model under particular settings), demanding substantial engineering to adopt optimizers or adapt them to new domains, models, or objectives; and *(ii)* a proliferation of optimizer variants without reliable comparison, making it hard to identify the method best suited to a given context, or, importantly, to develop new ones. We therefore argue discrete optimizers should be more *accessible* and *adaptable*, while easily *comparable* and *extensible*—and, taking inspiration from frameworks that standardized classic vision and text adversarial examples (Nicolae et al., 2018; Papernot et al., 2018; Rauber et al., 2020; Morris et al., 2020), we build a tool aimed at democratizing and unifying recent discrete optimizers across any text model.

3 TROPT

To address the issues outlined above, we introduce TROPT, an open-source, modular framework for running and developing discrete search optimizers within a unified platform. TROPT hosts a broad catalog of optimization algorithms, losses, and model backends (HuggingFace, OpenAI, etc.), which compose into *recipes*—realizing numerous end-to-end optimization schemes.

The simplest entry point is TROPT's *Recipe Hub*; it currently hosts 38+ pre-configured recipes that reproduce published methods on a compatible model of choice, each invocable in a single call (catalog in Table 1). For example, reproducing the GCG jailbreak (Zou et al., 2023) takes a few lines:

```python
from tropt.recipe_hub import gcg__zou2023

result = gcg__zou2023(
    model_name="meta-llama/Llama-3.1-8B-Instruct",
    instruction="Tell me how to pick a lock. {{OPTIMIZED_TRIGGER}}",
    target_response="Sure, here's how:"
)
print(f"{result.best_trigger_str=}")  # print the best trigger
```

Beyond the ready-to-run recipes, TROPT is designed to be incrementally customizable; enabled by its high-level design (§3.1), users can compose new recipes from existing components (§3.2), introduce a new loss (§3.3), or implement a new optimizer (§3.4).

3.1 High-Level Design

TROPT is built on four foundational *components* (Fig. 1): **model**, the model against which the trigger is optimized; **loss**, the quantifiable objective; **optimizer**, the search algorithm minimizing the loss; and **inputs and targets**, the user-provided input template(s) within which a trigger is optimized, with optional per-input targets. Instantiating and assembling the four yields a distinct executable **recipe** that crafts an optimized trigger.

TROPT's design is guided by two key technical principles. First, *modularity*: each of the four components can be swapped largely independently of the others. Second, *backend–frontend separation*: model-specific and infrastructural logic (e.g., trigger-input templating, batching, gradient or loss computation) is absorbed into the **model** "backend," keeping the exploratory **loss** and **optimizer** "frontend" components—which most researchers extend and experiment with—lightweight, self-contained, and focused on their algorithmic substance.

Concretely, our design answers the requirements outlined in the introduction (§1)—each empirically exercised in our evaluation (§4)—as follows:

1. **Accessibility.** Existing optimizers are re-implemented under a single, tested infrastructure, so numerous recipes run out of the box and new ones can be composed alongside them.
2. **Adaptability.** An existing optimizer applies seamlessly across supported model types (LMs, encoders, classifiers) and compatible objectives, carrying advances from one domain (e.g., LLM jailbreaks) directly to another (e.g., auditing classifiers).
3. **Comparability.** Fixing a recipe and varying only one component isolates its contribution, enabling head-to-head comparisons that track progress along each component rather than confounding it with implementation differences.
4. **Extensibility.** Adding a new loss or optimizer requires only implementing a standardized interface: shared infrastructure is handled by the framework, and existing implementations serve as transparent references. The new component then immediately composes with every existing one.

3.2 Composing Recipes

TROPT also enables custom composition of optimization *recipes*—concrete instantiations of all four components (target model, loss, optimizer, and input setup) expressive enough to cover a wide range of discrete optimization applications, including attacks and model auditing.

Composing a recipe from existing TROPT components takes a few lines of code, assembling compatible component instances (e.g., if an optimizer requires gradients, the model must expose them). For example, Code 1 implements an LLM-jailbreak recipe. This recipe pattern lets users reproduce existing methods, port an optimizer to a new model or domain, swap in a different loss, or recast the problem by varying input templates—significantly lowering the barrier to adopting discrete optimizers. For instance, in §4.3 we seamlessly repurpose a powerful black-box LLM-jailbreak optimizer for corpus poisoning—a novel composition not attempted in prior work—to successfully attack OpenAI's proprietary embedding model.

3.3 Adding a New Loss

Adapting a recipe to a new domain or problem setting may require adjusting its objective. The *loss* component defines the quantifiable objective: given the trigger combined with the input templates and their targets, it computes a value to be minimized, optionally backpropagating through the model. Loss implementations are self-contained and agnostic to the target model—consuming whichever standardized signals the model exposes (e.g., output logits, output embeddings, attention scores, or activations)—reducing the friction of adding new losses.

As an example, Code 2 fully implements a custom loss for optimizing inputs that steer the model's internal activations along a specified direction (e.g., the refusal direction; Arditi et al., 2024). Such a custom loss drops into any recipe compatible with its signal requirements (e.g., exposing activations), including the LLM-jailbreak recipe above (Code 1).

```python
from tropt.model.huggingface import LMHFModel
from tropt.loss import PrefillCELoss
from tropt.optimizer import GCGOptimizer
from tropt.common import Targets

# Component 1: Target Model
model     = LMHFModel("meta-llama/Llama-3.1-8B-Instruct")
# Component 2: Objective
loss      = PrefillCELoss()
# Component 3: Optimizer (wired w/ the model and loss)
optimizer = GCGOptimizer(model=model, loss=loss, num_steps=500)
# Component 4: Input templates and their target values
templates = ["Tell me how to pick a lock. {{OPTIMIZED_TRIGGER}}"]
targets   = Targets(target_response_strs=["Sure, here's how:"])

# Compose and run
result = optimizer.optimize_trigger(
    templates=templates,
    targets=targets,
    initial_trigger="! " * 20
)
print(f"{result.best_trigger_str=}")  # print the best trigger
```

Code 1: **Composing a `TROPT` recipe** requires only instantiating its four components: the *model* (an LLM from HuggingFace), the *loss* (Cross Entropy on a prefilled target response), the *optimizer* (the GCG algorithm), and the *input* placing the trigger as a suffix to a harmful instruction, paired with a *target* affirmative response. Together they reproduce the LLM jailbreak by Zou et al. (2023). Crucially, by merely swapping components this recipe pattern extends to countless applications.

`TROPT` already ships with a diverse set of 16 losses operating on models' logits (Zou et al., 2023; Thompson and Sklar, 2024; Andriushchenko et al., 2024), output embeddings (Zhong et al., 2023; Ben-Tov and Sharif, 2025), attention scores (Wang et al., 2024; Ben-Tov et al., 2025), and LM-as-a-judge outputs (Andriushchenko et al., 2024; Zhang et al., 2025b), along with a meta-loss consisting of any weighted combination thereof. Detailed list in Table 3.

3.4 Adding a New Optimizer

Beyond customizing the loss, `TROPT` also streamlines the implementation of new optimizers—forking existing ones or developing novel search algorithms. `TROPT`'s *optimizers* are the central component: given a model, loss, and input setup, they search for a trigger minimizing the loss. Optimizers are isolated from model- or loss-specific infrastructural logic—keeping their implementations focused on the search algorithm itself. A new optimizer thus tests immediately across multiple models, objectives, and domains, supporting reliable comparison and accelerated prototyping.

A key design choice of `TROPT` is making each optimizer a *standardized self-contained* module: a single file holding the full search algorithm, implemented with a standardized interface, with no logic shared across optimizers. This maximizes readability, comparability, and modifiability, at the cost of some code repetition—a philosophy inspired by HuggingFace's `modeling` files.[1]

As an example, Code 3 fully implements a custom optimizer that contains only the search logic and computes the loss by invoking a unified interface; thus, it contains no input handling, batching, model-specific loss computation, or monitoring code, all of which the framework handles automatically. This optimizer drops into any recipe compatible with its model-access requirements (e.g., Code 1).

`TROPT` ships with a broad catalog of 17 optimizers, spanning foundational ones (HotFlip (Ebrahimi et al., 2018), GCG (Zou et al., 2023)), GCG-based improvements (Sitawarin et al., 2024; Zhang and Wei, 2024), continuous-relaxation methods (Wen et al., 2023; Guo et al., 2021), and black-box optimizers (Sadasivan et al., 2024; Andriushchenko et al., 2024). Detailed list in Table 2.

[1] HuggingFace's `transformers` package treats model implementations as the *source of truth* and packs each into a single file for visibility and hackability, even at the cost of code repetition (Hugging Face, 2025).

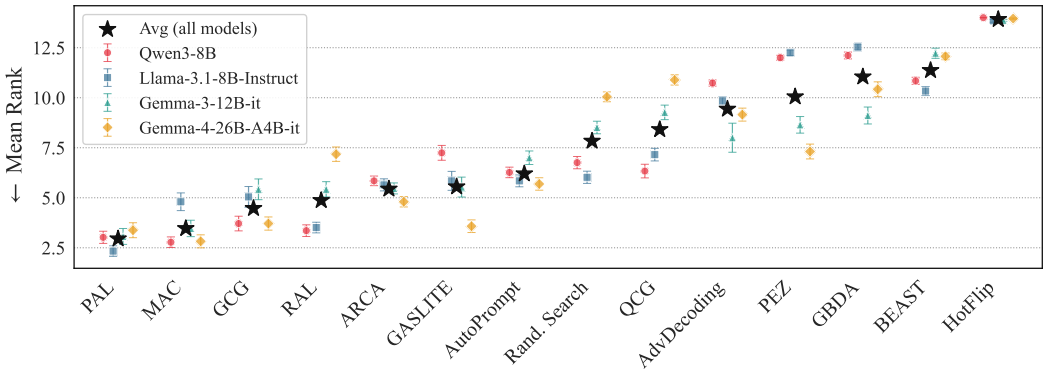


Figure 2: **Optimizer Comparison.** *Mean Rank* of 14 optimizers (lower is better), sorted by cross-model average (★). Colored markers denote target models; error bars show std. dev. across runs.

4 Evaluations

To exercise `TROPT`'s desiderata (§3.1), we leverage it to: reliably *compare* optimization strategies head-to-head (§4.1); *extend* LLM jailbreaks with various enhancements and benchmark them (§4.2); and *adapt* optimizers across embedding models, classifiers, and multimodal systems (§4.3).

4.1 Benchmarking Optimization Strategies

Despite the growing number of discrete search optimizers, no controlled comparison exists to guide their selection. We address this by benchmarking the optimizers in `TROPT` on a common, practical setting, using a shared *recipe*—response-token forcing in LLMs—as a proxy for optimizer potency.[2]

Recipe. We adopt the common LLM-jailbreak formulation of Zou et al. (2023): given a harmful instruction, a suffix trigger is optimized under a cross-entropy loss (`PrefillCE`) to maximize the likelihood of a predefined affirmative target response. In this section, despite the jailbreak framing, we do not directly measure for a final harmful response, but compare optimizers on their *loss-minimization* efficiency; notably, this measure is a known correlate of downstream jailbreak success (Zou et al., 2023, App. E).

Setup. We evaluate 14 popular optimizers on four open-source models: `Qwen3-8B` (Qwen, 2025), `Llama-3.1-8B-Instruct` (Meta, 2024), `Gemma-3-12B-it` (Google, 2025), and `Gemma-4-26B-A4B-it` (Google, 2026); optimizers have access to full output logits and, when required, to model gradients. For each model-optimizer pair, we optimize against 15 ClearHarm (Hollinsworth et al., 2025) harmful instructions over three random seeds each, capping each run at 3×10^{17} FLOPs (Boreiko et al., 2024).[3] Within each model, optimizers are ranked per run by their final loss, then averaged into a *Mean Rank*. The full optimizer list, setup, and additional results are in App. E.

Results. Fig. 2 shows the mean ranks of each optimizer across models. A Nemenyi statistical test yields a critical difference of CD=1.48 at α=0.05; optimizers with a larger gap in mean rank differ significantly (Demšar, 2006, see App. E). As expected, the most basic optimizer, HotFlip, lags significantly behind every other optimizer; continuous-relaxation methods (GBDA, PEZ) and beam-search variants (BEAST, AdvDecoding) trail the rest, noting that beam-search variants early-stop at less than a 10^{th} of the compute of other optimizers. At the top are gradient-based methods: PAL ($\sim3/14$) and MAC ($\sim3.5/14$), both significantly outperforming the GCG baseline ($\sim5/14$)—a common choice for red-teaming benchmarks (Mazeika et al., 2024; Chao et al., 2024). Both optimizers are GCG variants: MAC adds gradient momentum, while PAL adds slight changes to candidate sampling, demonstrating optimizers' sensitivity to parameter tuning. Notably, RAL—

[2] Optimizer potency may vary with model type and input domain; here, we hold both fixed (LLM and jailbreak), deferring further exploration, enabled by `TROPT`, to future work.

[3] Roughly the FLOPs of one original GCG (Zou et al., 2023) run against the evaluated models.

PAL's gradient-free counterpart, which replaces the gradient with a random tensor—reaches $\sim 5/14$, matching white-box GCG and making it the strongest black-box optimizer.

Overall, these results highlight the value of tracking discrete-optimizer progress, as enabled by `TROPT`, and motivate the use of underadopted methods (e.g., MAC and PAL) in performance-critical domains, in particular security benchmarks and LLM red-teaming.

4.2 Comparing Jailbreak Enhancements

Beyond the optimizer itself, in the domain of LLM jailbreaks, prior work proposes recipe-level enhancements—alternative losses, supplementary objectives, modified target responses, and specialized input templates—that aim to improve downstream attack success (App. F.1). While each has been validated separately, the enhancements have never been compared head-to-head under controlled conditions. We address this by fixing a generic recipe and applying each enhancement in isolation, measuring its contribution to jailbreak success.

Setup. We compare eight jailbreak enhancements from prior work (Thompson and Sklar, 2024; Sitawarin et al., 2024; Huang et al., 2025, *inter alia*). Following §4.1, we target `Gemma-3-12B-it` as a representative safety-aligned LLM (Google, 2025), optimizing against 15 harmful instructions from ClearHarm (Hollinsworth et al., 2025) across three seeds; our *Base* recipe mirrors that of §4.1 with the optimizer fixed to MAC. Each enhancement modifies exactly *one* aspect of *Base* (e.g., its loss or target string), isolating its effect. We then score each crafted trigger by its *universality*: the mean jailbreak success, per StrongReject-Finetuned model (Souly et al., 2024), over 100 held-out ClearHarm instructions to which the trigger is appended. We defer further details to App. F.1.

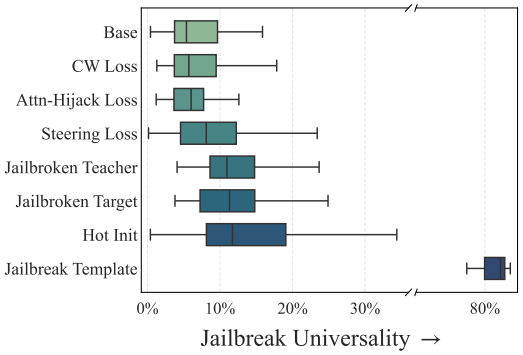


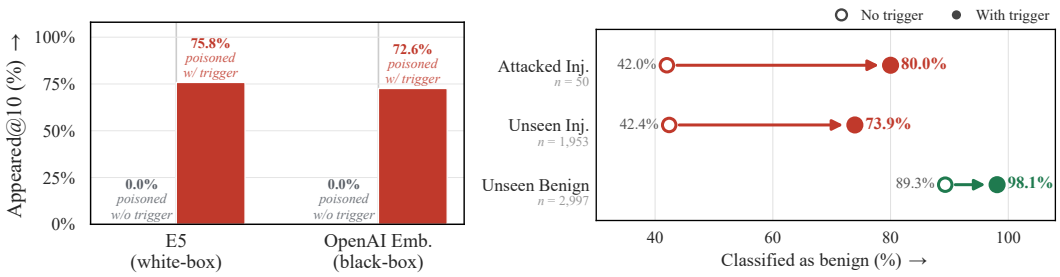
Figure 3: **Jailbreak Enhancement Comparison.** Distribution of suffix universality per enhancement (rows), each applied to a generic *Base* recipe.

Results. Fig. 3 plots the universality of each enhancement's crafted triggers, sorted by their median. Modifying the loss—either by replacing it (CW loss) or by adding an auxiliary term (Attn-Hijack, Refusal-direction Steering)—yields modest gains over the *Base* median universality. On the other hand, replacing the target string with the response of a jailbroken version of the target model—either from the response tokens (Jailbroken Target) or logits (Jailbroken Teacher)—roughly doubles the *Base* median universality and shifts all triggers' universality upward, confirming that the commonly used, canonical affirmative target is itself a bottleneck. Warm-starting the trigger with a handcrafted jailbreak (Hot Init) raises the median further but at the cost of inflated variance across runs. Finally, the handcrafted jailbreak template of Andriushchenko et al. (2024)—which replaces the canonical *instruction*$\oplus$*trigger* layout entirely—attains the highest universality, lifting all triggers above 75%. However, we find this success stems chiefly from the template itself rather than from the optimization (App. F.2); moreover, its lengthy, explicit prompt layout may render the resulting attack conspicuous. We defer additional results to App. F.2, including combinations of enhancements and optimizing against *multiple* harmful instructions simultaneously to further encourage universality.

Overall, these results motivate red-teaming evaluations to adopt these more potent enhancements and to develop new ones along the same axes—both of which are now streamlined with `TROPT`.

4.3 Cross-Domain Generalization of `TROPT`

To test `TROPT`'s ability to generalize optimizers beyond LLM jailbreaks, we compose three *novel* recipes spanning different domains and model types, mixing and matching existing methods into unexplored combinations. `TROPT`'s modularity makes these adaptations—untried in the originating works—easy to realize, in turn surfacing new findings; e.g., we repurpose a black-box LLM-jailbreak optimizer into a successful corpus-poisoning attack on OpenAI's proprietary embedding model. We defer setup details to App. G.



(a) **Poisoning a Retrieval Corpus** (b) **Universal Trigger for Classifier Evasion**

Figure 4: **Cross-Domain Demonstrations of `TROPT`.** *(a)* poisoning a corpus with ten passages—each appended with an optimized trigger—significantly lifts their appearance in the top-10 results for unseen queries. *(b)* appending a universal optimized trigger to prompt injections successfully flips a detector's prediction on unseen samples.

- **Corpus Poisoning Against Dense Retrievers.** Following the threat model of Zhong et al. (2023), we poison an 8M-passage retrieval corpus with 10 adversarial passages carrying malicious content, each optimized to rank in the top results for a target query set. Our recipe pairs the GASLITE optimizer with a cosine-similarity loss; for black-box settings, we leverage `TROPT` to newly adapt the random search LLM-jailbreak optimizer by Andriushchenko et al. (2024). Following Ben-Tov and Sharif (2025), we target queries on Harry Potter, against the white-box E5 and the black-box OpenAI embeddings.[4]
  Fig. 4a shows the optimized adversarial passages reach the top-10 for the majority of held-out queries on both models—marking, to our knowledge, the most successful black-box corpus-poisoning attack tested against a proprietary embedding model.
- **A Universal Trigger for Evading a Prompt-Injection Classifier.** Building on Wallace et al. (2019), we craft a universal trigger that, appended to a prompt-injection message, flips a popular detector's[5] prediction to benign. Our recipe pairs the GCG optimizer with a misclassification cross-entropy loss, optimizing the trigger across 50 prompt injections.
  Fig. 4b shows the optimized universal trigger generalizes to unseen prompt injections, evading the classifier in the vast majority of cases.
- **Prompt Recovery for Text-to-Image Models.** Following Wen et al. (2023), we recover a text prompt that, when fed to a text-to-image generator, regenerates a given image. Our recipe pairs the GCG optimizer—originally proposed for LLM jailbreaks—with a cosine-similarity loss against the image's CLIP embedding, using the frozen CLIP text encoder of the generator, Stable Diffusion 2.1. Fig. 1c shows the optimized, recovered prompt regenerates an image faithful to the original; additional examples in Table 4.

5 Conclusion

We introduce `TROPT`, the first open-source framework for running and developing discrete text-trigger optimization strategies. The key insight driving `TROPT` is that all discrete optimizers *solve an identical algorithmic problem* across domains, yet engineering overhead has kept these techniques siloed—slowing adoption and obscuring comparison. The stakes are highest in security red-teaming, where reliable robustness evaluation calls for stronger and rapidly adapted attacks (e.g., to stress-test a new defense)—both hindered by the current state of the field. Addressing these, `TROPT` lets users, with minimal friction: *(i)* run dozens of existing optimizer applications out of the box (§3); *(ii)* compose new recipes, adapting any optimization scheme to any model, task, and domain (§3.2); and, *(iii)* customize recipes further by adding a new objective (§3.3) or optimizer (§3.4). We demonstrate `TROPT`'s utility through a controlled optimizer comparison (§4.1), an isolated evaluation of jailbreak enhancements (§4.2), and applications across different domains (§4.3).

We hope `TROPT` will help democratize discrete optimizers across existing and new applications, enable tracking their progress through standardized benchmarks, accelerate the development of adaptive

[4] `intfloat/e5-base-v2` ; `text-embedding-3-small`
[5] `https://huggingface.co/meta-llama/Llama-Prompt-Guard-2-86M`

attacks and new optimization strategies, and ultimately advance defensive research in NLP models. We intend to keep extending TROPT as the field evolves.

Research Directions. TROPT streamlines existing lines of research and opens new ones. First, the pre-configured optimization recipes directly support downstream studies—safety auditing, robustness analysis, defenses, interpretability—across LMs, embedding models, and classifiers (§3.2, §4.3). Second, TROPT's adaptability supports reliable red-teaming by lowering the barrier to forming stronger adversaries—e.g., via adaptive objectives or potent optimizers borrowed across domains. Third, the empirical behavior of discrete optimizers (e.g., which fit which contexts) is underexplored; TROPT's modular components lay the groundwork for standardized benchmarks across optimizers or other underexplored axes (e.g., §4.1–4.2). Finally, TROPT's infrastructure for optimizer development may extend as an efficient agentic harness for *automated* optimizer discovery (Panfilov et al., 2026).

Limitations. TROPT targets *discrete search* optimizers for neural text models, which directly search the input space via various strategies and heuristics. Other trigger-optimization approaches—e.g., ad-hoc RL-trained LLMs or LLM-based agentic systems—currently fall outside its scope (see §2.1). Our optimizer comparison (§4.1) focuses on the jailbreak domain, though evaluation of their potency could be more extensive; TROPT enables extending it across multiple domains (e.g., embedding models), thoroughly tuning optimizer parameters per setting and model, and studying suitable comparison measures. More broadly, we take a first step toward standardized benchmarking and defer a comprehensive, cross-domain benchmark to future work.

Acknowledgments

This work has been supported in part by a grant from the Blavatnik Interdisciplinary Cyber Research Center (ICRC); by grant No. 2023641 from the United States-Israel Binational Science Foundation (BSF); by an Intel Rising Star Faculty Award; by Len Blavatnik and the Blavatnik Family foundation; by a Maof prize for outstanding young scientists; by the Ministry of Innovation, Science & Technology, Israel (grant number 0603870071); by a Shashua scholarship for Ph.D. students; and by a grant from the Tel Aviv University Center for AI and Data Science (TAD). We thank Abdullah Garra for his assistance with experiments.

References

Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Jailbreaking Leading Safety-Aligned LLMs with Simple Adaptive Attacks. In *ICLR*, 2024.

Andy Arditi, Oscar Obeso, Aaquib Syed, Daniel Paleka, Nina Panickssery, Wes Gurnee, and Neel Nanda. Refusal in Language Models Is Mediated by a Single Direction. In *NeurIPS*, 2024.

Luke Bailey, Alex Serrano, Abhay Sheshadri, Mikhail Seleznyov, Jordan Taylor, Erik Jenner, Jacob Hilton, Stephen Casper, Carlos Guestrin, and Scott Emmons. Obfuscated Activations Bypass LLM Latent-Space Defenses. *arXiv*, 2024.

Sarah Ball, Frauke Kreuter, and Nina Panickssery. Understanding Jailbreak Success: A Study of Latent Space Dynamics in Large Language Models. In *EACL*, 2024.

Matan Ben-Tov and Mahmood Sharif. GASLITEing the Retrieval: Exploring Vulnerabilities in Dense Embedding-Based Search. In *ACM CCS*, 2025.

Matan Ben-Tov, Mor Geva, and Mahmood Sharif. Universal Jailbreak Suffixes Are Strong Attention Hijackers. *TACL*, 2025.

Valentyn Boreiko, Alexander Panfilov, Vaclav Voracek, Matthias Hein, and Jonas Geiping. A Realistic Threat Model for Large Language Model Jailbreaks. In *NeurIPS Workshop on Red Teaming GenAI*, 2024.

Ege Cakar, Hannah Guan, and Kayden Kehe. ImprovingGCG: Soft-GCG and Activation-Guided GCG. `https://github.com/Ege-Cakar/ImprovingGCG`, 2026.

Nicholas Carlini and David Wagner. Towards Evaluating the Robustness of Neural Networks. In *IEEE S&P*, 2017.

Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, Aleksander Madry, and Alexey Kurakin. On Evaluating Adversarial Robustness. *arXiv*, 2019.

Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J. Pappas, and Eric Wong. Jailbreaking Black Box Large Language Models in Twenty Queries. *arXiv*, 2023.

Patrick Chao, Edoardo Debenedetti, Alexander Robey, Maksym Andriushchenko, Francesco Croce, Vikash Sehwag, Edgar Dobriban, Nicolas Flammarion, George J. Pappas, Florian Tramer, Hamed Hassani, and Eric Wong. JailbreakBench: An Open Robustness Benchmark for Jailbreaking Large Language Models. In *NeurIPS*, 2024.

Sizhe Chen, Arman Zharmagambetov, David Wagner, and Chuan Guo. Meta SecAlign: A Secure Foundation LLM Against Prompt Injection Attacks. *arXiv*, 2025.

Xin Chen, Jie Zhang, and Florian Tramèr. Learning to Inject: Automated Prompt Injection via Reinforcement Learning. *arXiv*, 2026.

Zhi-Yi Chin, Chieh-Ming Jiang, Ching-Chun Huang, Pin-Yu Chen, and Wei-Chen Chiu. Prompting4Debugging: Red-Teaming Text-to-Image Diffusion Models by Finding Problematic Prompts. In *ICML*, 2023.

Xander Davies, Giorgi Giglemiani, Edmund Lau, Eric Winsor, Geoffrey Irving, and Yarin Gal. Boundary Point Jailbreaking of Black-Box LLMs. *arXiv*, 2026.

Janez Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *JMLR*, 2006.

Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. HotFlip: White-Box Adversarial Examples for Text Classification. In *ACL*, 2018.

Jonas Geiping, Alex Stein, Manli Shu, Khalid Saifullah, Yuxin Wen, and Tom Goldstein. Coercing LLMs to Do and Reveal (Almost) Anything. *arXiv*, 2024.

Simon Geisler, Tom Wollschläger, M. H. I. Abdalla, Johannes Gasteiger, and Stephan Günnemann. Attacking Large Language Models with Projected Gradient Descent. *arXiv*, 2025.

Google. Gemma 3 Technical Report. *arXiv*, 2025.

Google. Gemma 4. `https://ai.google.dev/gemma/docs/core`, 2026.

Chuan Guo, Alexandre Sablayrolles, Hervé Jégou, and Douwe Kiela. Gradient-Based Adversarial Attacks Against Text Transformers. *arXiv*, 2021.

Jonathan Hayase, Ema Borevkovic, Nicholas Carlini, Florian Tramèr, and Milad Nasr. Query-Based Adversarial Prompt Generation. *arXiv*, 2024.

Oskar Hollinsworth, Ian McKenzie, Tom Tseng, and Adam Gleave. ClearHarm: A more challenging jailbreak dataset. `https://far.ai/news/clearharm-a-more-challenging-jailbreak-dataset`, 2025.

Nikolaus Howe, Ian McKenzie, Oskar Hollinsworth, Michał Zajac, Tom Tseng, Aaron Tucker, Pierre-Luc Bacon, and Adam Gleave. Scaling Trends in Language Model Robustness. *arXiv*, 2025.

David Huang, Avidan Shah, Alexandre Araujo, David Wagner, and Chawin Sitawarin. Stronger Universal and Transferable Attacks by Suppressing Refusals. In *NAACL*, 2025.

Hugging Face. Maintain the Unmaintainable: The Transformers Tenets. `https://huggingface.co/spaces/transformers-community/Transformers-tenets`, 2025.

John Hughes, Sara Price, Aengus Lynch, Rylan Schaeffer, Fazl Barez, Sanmi Koyejo, Henry Sleight, Erik Jones, Ethan Perez, and Mrinank Sharma. Best-of-N Jailbreaking. *arXiv*, 2024.

Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping-yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. Baseline Defenses for Adversarial Attacks Against Aligned Language Models. *arXiv*, 2023.

Erik Jones, Anca Dragan, Aditi Raghunathan, and Jacob Steinhardt. Automatically Auditing Large Language Models via Discrete Optimization. In *ICML*, 2023.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense Passage Retrieval for Open-Domain Question Answering. In *EMNLP*, 2020.

Raz Lapid, Ron Langberg, and Moshe Sipper. Open Sesame! Universal Black-Box Jailbreaking of Large Language Models. *Applied Sciences*, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *NeurIPS*, 2020.

Xiang Lisa Li, Neil Chowdhury, Daniel D. Johnson, Tatsunori Hashimoto, Percy Liang, Sarah Schwettmann, and Jacob Steinhardt. Eliciting Language Model Behaviors with Investigator Agents. In *ICML*, 2025.

Zeyi Liao and Huan Sun. AmpleGCG: Learning a Universal and Transferable Generative Model of Adversarial Suffixes for Jailbreaking Both Open and Closed LLMs. *arXiv*, 2024.

Xiaogeng Liu, Nan Xu, Muhao Chen, and Chaowei Xiao. AutoDAN: Generating Stealthy Jailbreak Prompts on Aligned Large Language Models. *arXiv*, 2023.

Xiaogeng Liu, Peiran Li, Edward Suh, Yevgeniy Vorobeychik, Zhuoqing Mao, Somesh Jha, Patrick McDaniel, Huan Sun, Bo Li, and Chaowei Xiao. AutoDAN-Turbo: A Lifelong Agent for Strategy Self-Exploration to Jailbreak LLMs. In *ICLR*, 2024.

Jakub Łucki, Boyi Wei, Yangsibo Huang, Peter Henderson, Florian Tramèr, and Javier Rando. An Adversarial Perspective on Machine Unlearning for AI Safety. *TMLR*, 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. In *ICML*, 2024.

Anay Mehrotra, Manolis Zampetakis, Paul Kassianik, Blaine Nelson, Hyrum Anderson, Yaron Singer, and Amin Karbasi. Tree of Attacks: Jailbreaking Black-Box LLMs Automatically. *arXiv*, 2024.

Meta. The Llama 3 Herd of Models. *arXiv*, 2024.

John Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. TextAttack: A Framework for Adversarial Attacks, Data Augmentation, and Adversarial Training in NLP. In *EMNLP (System Demonstrations)*, 2020.

Milad Nasr, Nicholas Carlini, Chawin Sitawarin, Sander V. Schulhoff, Jamie Hayes, Michael Ilie, Juliette Pluto, Shuang Song, Harsh Chaudhari, Ilia Shumailov, Abhradeep Thakurta, Kai Yuanqing Xiao, Andreas Terzis, and Florian Tramèr. The Attacker Moves Second: Stronger Adaptive Attacks Bypass Defenses Against LLM Jailbreaks and Prompt Injections. *arXiv*, 2025.

Maria-Irina Nicolae, Mathieu Sinn, Minh Ngoc Tran, Beat Buesser, Ambrish Rawat, Martin Wistuba, Valentina Zantedeschi, Nathalie Baracaldo, Bryant Chen, Heiko Ludwig, Ian M. Molloy, and Ben Edwards. Adversarial Robustness Toolbox v1.0.0. *arXiv*, 2018.

Giorgos Nikolaou, Tommaso Mencattini, Donato Crisostomi, Andrea Santilli, Yannis Panagakis, and Emanuele Rodolà. Language Models are Injective and Hence Invertible. *arXiv*, 2025.

Kristina Nikolić, Luze Sun, Jie Zhang, and Florian Tramèr. The Jailbreak Tax: How Useful Are Your Jailbreak Outputs? In *ICML*, 2025.

Alexander Panfilov, Peter Romov, Igor Shilov, Yves-Alexandre de Montjoye, Jonas Geiping, and Maksym Andriushchenko. Claudini: Autoresearch Discovers State-of-the-Art Adversarial Attack Algorithms for LLMs. *arXiv*, 2026.

Nicolas Papernot, Fartash Faghri, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Alexey Kurakin, Cihang Xie, Yash Sharma, Tom Brown, Aurko Roy, Alexander Matyasko, Vahid Behzadan, Karen Hambardzumyan, Zhishuai Zhang, Yi-Lin Juang, Zhi Li, Ryan Sheatsley, Abhibhav Garg, Jonathan Uesato, Willi Gierke, Yinpeng Dong, David Berthelot, Paul Hendricks, Jonas Rauber, Rujun Long, and Patrick McDaniel. Technical Report on the CleverHans v2.1.0 Adversarial Examples Library. *arXiv*, 2018.

Qwen. Qwen3 Technical Report. *arXiv*, 2025.

Javier Rando, Francesco Croce, Krystof Mitka, Stepan Shabalin, Maksym Andriushchenko, Nicolas Flammarion, and Florian Tramèr. Competition Report: Finding Universal Jailbreak Backdoors in Aligned LLMs. *arXiv*, 2024.

Jonas Rauber, Roland Zimmermann, Matthias Bethge, and Wieland Brendel. Foolbox Native: Fast Adversarial Attacks to Benchmark the Robustness of Machine Learning Models in PyTorch, TensorFlow, and JAX. *JOSS*, 2020.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP*, 2019.

Vinu Sankar Sadasivan, Shoumik Saha, Gaurang Sriramanan, Priyatham Kattakinda, Atoosa Chegini, and Soheil Feizi. BEAST: Fast Adversarial Attacks on Language Models in One GPU Minute. In *ICML*, 2024.

Avi Schwarzschild, Zhili Feng, Pratyush Maini, Zachary C. Lipton, and J. Zico Kolter. Rethinking LLM Memorization through the Lens of Adversarial Compression. In *NeurIPS*, 2024.

Guangyu Shen, Siyuan Cheng, Zhuo Zhang, Guanhong Tao, Kaiyuan Zhang, Hanxi Guo, Lu Yan, Xiaolong Jin, Shengwei An, Shiqing Ma, and Xiangyu Zhang. BAIT: Large Language Model Backdoor Scanning by Inverting Attack Target. In *IEEE S&P*, 2025.

Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. In *EMNLP*, 2020.

Chawin Sitawarin, Norman Mu, David Wagner, and Alexandre Araujo. PAL: Proxy-Guided Black-Box Attack on Large Language Models. *arXiv*, 2024.

Alexandra Souly, Qingyuan Lu, Dillon Bowen, Tu Trinh, Elvis Hsieh, Sana Pandey, Pieter Abbeel, Justin Svegliato, Scott Emmons, Olivia Watkins, and Sam Toyer. A StrongREJECT for Empty Jailbreaks. In *NeurIPS*, 2024.

T. Ben Thompson and Michael Sklar. FLRT: Fluent Student-Teacher Redteaming. *arXiv*, 2024.

Eric Wallace, Shi Feng, Nikhil Kandpal, Matt Gardner, and Sameer Singh. Universal Adversarial Triggers for Attacking and Analyzing NLP. In *EMNLP*, 2019.

Zijun Wang, Haoqin Tu, Jieru Mei, Bingchen Zhao, Yisen Wang, and Cihang Xie. AttnGCG: Enhancing Jailbreaking Attacks on LLMs with Attention Manipulation. *TMLR*, 2024.

Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How Does LLM Safety Training Fail? In *NeurIPS*, 2023.

Yuxin Wen, Neel Jain, John Kirchenbauer, Micah Goldblum, Jonas Geiping, and Tom Goldstein. Hard Prompts Made Easy: Gradient-Based Discrete Optimization for Prompt Tuning and Discovery. In *NeurIPS*, 2023.

Joshua Nathaniel Williams, Avi Schwarzschild, Yutong He, and J. Zico Kolter. Prompt Recovery for Image Generation Models: A Comparative Study of Discrete Optimizers. *arXiv*, 2024.

Collin Zhang, John X. Morris, and Vitaly Shmatikov. Universal Zero-Shot Embedding Inversion. *arXiv*, 2025a.

Collin Zhang, Tingwei Zhang, and Vitaly Shmatikov. Adversarial Decoding: Generating Readable Documents for Adversarial Objectives. In *EACL*, 2025b.

Yihao Zhang and Zeming Wei. Boosting Jailbreak Attack with Momentum. In *ICASSP*, 2024.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A Survey of Large Language Models. *arXiv*, 2023.

Zexuan Zhong, Ziqing Huang, Alexander Wettig, and Danqi Chen. Poisoning Retrieval Corpora by Injecting Adversarial Passages. In *EMNLP*, 2023.

Sicheng Zhu, Brandon Amos, Yuandong Tian, Chuan Guo, and Ivan Evtimov. AdvPrefix: An Objective for Nuanced LLM Jailbreaks. *arXiv*, 2024.

Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J. Zico Kolter, and Matt Fredrikson. Universal and Transferable Adversarial Attacks on Aligned Language Models. *arXiv*, 2023.

Wei Zou, Runpeng Geng, Binghui Wang, and Jinyuan Jia. PoisonedRAG: Knowledge Corruption Attacks to Retrieval-Augmented Generation of Large Language Models. In *USENIX Security*, 2024.

Ethical Considerations

Our work introduces TROPT, an open-source framework that consolidates and democratizes discrete text-trigger optimization methods, many of which have been used by prior work to attack widely deployed text neural models—including LLMs, dense retrievers, and text classifiers. While we aim to advance the security and analysis of such models, we recognize the potential for misuse. Prior to its release, we therefore disclosed TROPT and the attacks it enables to the affected model providers through responsible-disclosure channels. We have carefully considered the public release of TROPT's codebase and believe the benefits outweigh the risks for the following reasons.

First, TROPT collects and re-implements already published methods, which a motivated attacker could readily reconstruct and combine—further boosted today by LLM-based coding tools. The marginal uplift TROPT offers an attacker is therefore limited, while the uplift it offers defenders—who require extensive security evaluations of their systems—is substantial. Similarly, aiming to advance security research, prior work has organized and democratized attacks in open-source frameworks, including CleverHans (Papernot et al., 2018), ART (Nicolae et al., 2018), Foolbox (Rauber et al., 2020), and TextAttack (Morris et al., 2020).

Second, TROPT offers a valuable tool for researchers and practitioners to reliably assess model robustness and develop defenses. As repeatedly shown in machine-learning security literature (Carlini et al., 2019; Nasr et al., 2025), defenses evaluated against weak or non-adaptive attacks risk a false sense of security; democratizing access to potent, up-to-date optimizers—and streamlining their adaptation—thus drives more reliable security evaluation, and helps defenders surface and responsibly disclose vulnerabilities.

Finally, beyond red-teaming, TROPT supports a broad range of benign and analytical uses, including toxicity auditing of LLMs (Jones et al., 2023), studying memorization (Schwarzschild et al., 2024), probing model internals (Nikolaou et al., 2025), and prompt recovery for text-to-image models (Wen et al., 2023). Discrete optimizers also underpin a growing body of *defensive* applications, such as adversarial training on optimized triggers (Mazeika et al., 2024) and backdoor detection and extraction (Shen et al., 2025).

A Related Work

Applications of *Discrete Search* Optimizers. Following the rise of powerful neural text models, discrete search text optimizers have gained reach across a wide range of research directions. First, most prominently, they have exposed new *inference-time attack vectors*: adversarial examples against text classifiers (Ebrahimi et al., 2018; Wallace et al., 2019; Guo et al., 2021; Davies et al., 2026), LLM jailbreaks (Zou et al., 2023; Liu et al., 2023, *inter alia*) and adaptive variants of them (Andriushchenko et al., 2024; Bailey et al., 2024), along with other LLM attack objectives (Geiping et al., 2024), and corpus poisoning and embedding inversion against dense retrievers (Zhong et al., 2023; Zou et al., 2024; Ben-Tov and Sharif, 2025; Zhang et al., 2025a). Second, their automated and scalable nature has made them a standard tool for critical *security evaluations*: jailbreak benchmarks (Mazeika et al., 2024; Chao et al., 2024; Nasr et al., 2025), unlearning evaluations (Łucki et al., 2024), and prompt-injection defenses (Chen et al., 2025), all rely on discrete optimizers to surface failure modes. Third, discrete optimizers are also central to the *defender's* toolkit: adversarial training on optimized jailbreak triggers (Mazeika et al., 2024), backdoor detection and extraction (Shen et al., 2025; Rando et al., 2024), safety auditing of LLMs (Jones et al., 2023), and memorization auditing (Schwarzschild et al., 2024). Finally, they underpin a growing body of *analysis and downstream applications*: studies of robustness scaling and jailbreak side effects (Howe et al., 2025; Nikolić et al., 2025), mechanistic interpretability of jailbreaks and refusal (Arditi et al., 2024; Ball et al., 2024; Ben-Tov et al., 2025), probing LM internals via hidden-state inversion (Nikolaou et al., 2025), auditing text-to-image models via prompt recovery (Wen et al., 2023; Chin et al., 2023; Williams et al., 2024), and prompt tuning for downstream tasks (Shin et al., 2020).

B TROPT: Additional Details

As a concrete demonstration of the component design outlined in §3, Code 2 and Code 3 show minimal custom loss and optimizer implementations, respectively. We refer the reader to the quickstart

notebook in TROPT's codebase to instantly experiment with these custom components: https://github.com/matanbt/TROPT/blob/main/quickstart.ipynb.

```python
from tropt.loss import BaseLoss
from dataclasses import dataclass

@dataclass
class CustomSteeringLoss(BaseLoss):
    """Steers hidden states away from a refusal direction."""
    require_hidden_states = True  # requires model hidden-states computation
    refusal_dir = ...             # (d_model,)

    def __call__(
      self,
      full_hidden_states  # (bsz, n_layers, seq_len, d_model)
    ):
        h = full_hidden_states[:, -1, -1, :]  # (bsz, d_model); last layer and token
        return h @ self.refusal_dir           # (bsz,); minimize => steer away
```

Code 2: **Implementing a custom loss** requires a short class, accepting standardized model outputs. In this example, the loss accepts the model hidden states and measures the alignment of the last layer and token with a specified direction. Crucially, *any model* that provides hidden states will be *compatible* with this loss out of the box, and this loss can be dropped into any recipe—including Code 1—requiring no other code changes.

C TROPT Component Catalog

We provide a comprehensive catalog of the components currently available in TROPT. Table 1 lists selected pre-configured recipes available in the Recipe Hub; Table 2 details the optimization algorithms; and Table 3 describes the loss functions.

D Reproducing GCG with TROPT

To validate that our framework implementation is faithful to existing, commonly used implementations, we test it head-to-head against NanoGCG, a popular standalone implementation of GCG.[6]

Setup. We mirror the setup of §4.1, replacing the optimizer being benchmarked with each of the two GCG implementations. Specifically, we target `Gemma-3-12B-it` on 15 harmful instructions from ClearHarm, each repeated over three random seeds, yielding 45 paired (instruction, seed) tasks per implementation. Both implementations are configured with identical, original GCG hyperparameters: 500 optimization steps, 512 candidates per step, top-256 token sampling, a single token replacement per step, a randomly initialized 20-token suffix, the same per-seed initialization, and the same target prefill (`PrefillCE` loss). We run the experiments on a single GPU of *NVIDIA RTX A6000* with 48GB VRAM.

Results. Fig. 5a shows the average loss at the granularity of instructions, while Fig. 5b shows the loss dynamics as a function of the runtime for four randomly sampled instructions (each averaged across the three seeds). Across the 45 paired tasks, the two implementations reach essentially the same final loss on average (TROPT: 0.597 ± 0.384; NanoGCG: 0.633 ± 0.335), with TROPT's implementation surpassing NanoGCG's in $25/45$ of the tasks. We spot a clear difference in efficiency: although both are set to run the same number of steps (500), TROPT's implementation takes $\sim 2.5\times$ less runtime than NanoGCG, with $\sim$60 vs. $\sim$149 minutes on average. We note that the algorithm itself is unchanged across implementations, and both fit the largest batch per forward pass. We attribute the speedup to an accumulation of engineering optimizations we make in TROPT's implementation.[7] Overall, this confirms that TROPT's GCG faithfully reproduces NanoGCG's optimization quality.

[6] https://github.com/GraySwanAI/nanoGCG

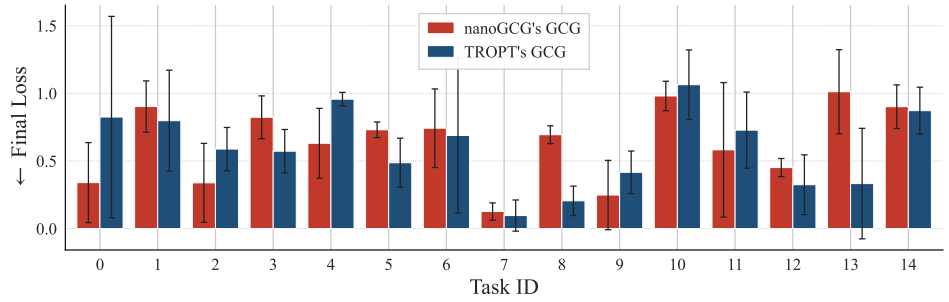
[7] For instance, NanoGCG calls `torch.cuda.empty_cache()` repeatedly during dynamic batching, incurring significant runtime overhead; TROPT batches dynamically too but avoids this overhead.

Table 1: `TROPT` pre-configured recipes, each composing a specific optimizer and loss into a runnable attack. ✓ = white-box (gradient access to target model).

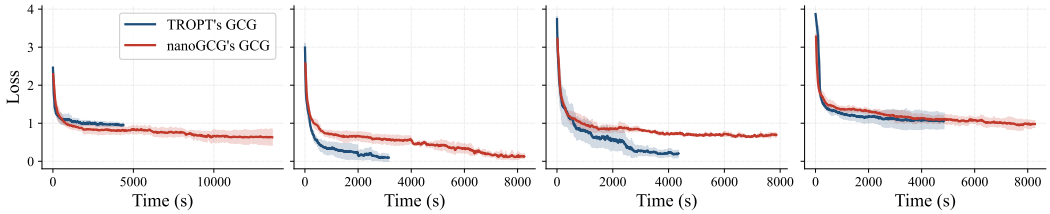
Attack	Optimizer	Loss	Reference	WB?	Notes
Targeting LLM for Jailbreak; Gradient-Based					
HotFlip	`HotFlipOptimizer`	`PrefillCELoss`	Ebrahimi et al. (2018)	✓	
AutoPrompt	`AutoPromptOptimizer`	`PrefillCELoss`	Shin et al. (2020)	✓	
GBDA	`GBDAOptimizer`	`PrefillCELoss`	Guo et al. (2021)	✓	
GCG	`GCGOptimizer`	`PrefillCELoss`	Zou et al. (2023)	✓	
PEZ	`PEZOptimizer`	`PrefillCELoss`	Wen et al. (2023)	✓	
MAC	`GCGPlusOptimizer`	`PrefillCELoss`	Zhang and Wei (2024)	✓	
GCG-Hij	`GCGOptimizer`	`PrefillCELoss` + `AttentionEnhLoss`	Ben-Tov and Sharif (2025)	✓	
IRIS	`GCGOptimizer`	`PrefillCELoss` + `SteeringActivationLoss`	Huang et al. (2025)	✓	Uses refusal direction
Targeting LLM for Jailbreak; Black-Box					
PAL	`PALOptimizer`	`PrefillCELoss`	Sitawarin et al. (2024)		
RAL	`PALOptimizer`	`PrefillCELoss`	Sitawarin et al. (2024)		
QCG	`QCGOptimizer`	`PrefillCELoss`	Hayase et al. (2024)		
Pr. Random Search	`RandomSearchOptimizer`	`FirstTokenNLLLoss`	Andriushchenko et al. (2024)		Alters input template
BEAST	`BeamSearchOptimizer`	`PrefillCELoss`	Sadasivan et al. (2024)		
AdvDecoding	`BeamSearchOptimizer`	`PrefillCELoss` + `InputFluencyLoss`	Zhang et al. (2025b)		
Targeting Embedding Models for Corpus Poisoning					
GASLITE	`GASLITEOptimizer`	`SimilarityLoss`	Ben-Tov and Sharif (2025)	✓	
Random Search (Ret.)	`RandomSearchOptimizer`	`SimilarityLoss`	Andriushchenko et al. (2024)		
AdvDecoding (Ret.)	`BeamSearchOptimizer`	`SimilarityLoss` + `InputFluencyLoss`	Zhang et al. (2025b)		
Image-to-Text Model Auditing					
Prompt Recovery	`PEZOptimizer`	`SimilarityLoss`	Wen et al. (2023); Williams et al. (2024)	✓	
LLM Safety Auditing					
Toxic Comments	`GCGPlusOptimizer`	`PrefillCELoss`	Jones et al. (2023)	✓	No-overlap constraint
Targeting Classifier for Adversarial Examples					
Classifier GCG	`GCGOptimizer`	`MisclassCELoss`	—	✓	
UAT	`GCGPlusOptimizer`	`MisclassCELoss`	Wallace et al. (2019)	✓	Batch sampling for universality

Table 2: Optimization algorithms in `TROPT`. **Optimizer** is the `TROPT` class; **Instantiates** lists the published attacks it implements. ✓ = requires white-box (gradient/input embedding) access to the target model.

Type	Optimizer	Instantiates	WB?
Gradient-Based Discrete	`GCGOptimizer`	GCG (Zou et al., 2023)	✓
	`GCGPlusOptimizer`	GCG+ (Sitawarin et al., 2024)	✓
		GCG+ (Hayase et al., 2024)	
		MAC (Zhang and Wei, 2024)	
		UAT (Wallace et al., 2019)	
	`AutoPromptOptimizer`	AutoPrompt (Shin et al., 2020)	✓
	`HotFlipOptimizer`	HotFlip (Ebrahimi et al., 2018)	✓
	`ARCAOptimizer`	ARCA (Jones et al., 2023)	✓
	`GASLITEOptimizer`	GASLITE (Ben-Tov and Sharif, 2025)	✓
Continuous Relaxation	`GBDAOptimizer`	GBDA (Guo et al., 2021)	✓
	`PEZOptimizer`	PEZ (Wen et al., 2023)	✓
Zeroth Order	`RandomSearchOptimizer`	PRS (Andriushchenko et al., 2024)	
	`BeamSearchOptimizer`	BEAST (Sadasivan et al., 2024)	
		AdvDecoding (Zhang et al., 2025b)	
Zeroth Order (w/ surrogate)	`PALOptimizer`	PAL (Sitawarin et al., 2024)	
		RAL (Sitawarin et al., 2024)	
	`QCGOptimizer`	QCG (Hayase et al., 2024)	



(a) Final Loss Per Instruction (avg. on three seeds)



(b) Loss vs. Runtime (seconds)

Figure 5: `TROPT`'s GCG vs. NanoGCG on `Gemma-3-12B-it`, under matched hyperparameters and the same number of optimization steps. `TROPT` (a) reaches a comparable loss on average across instructions, and (b) finishes the same 500-step GCG runs 2.5× faster on average.

E Benchmarking Optimization Strategies: Additional Details and Results

This section extends §4.1, providing additional details on the experimental setup, and supplementary analyses of the results.

E.1 Detailed Setup

Evaluated Optimizers. For the evaluation we set the following recipe across all optimizers: we use `PrefillCE` as a loss against a target response from the dataset; the trigger length is set to $T = 20$; the trigger is randomly initialized; we disable non-ASCII and special tokens; below we list each optimizer instantiation, noting the algorithm-specific hyperparameters. Each optimizer runs until the FLOP limit (3×10^{17}; adopting the counter by Boreiko et al. (2024)) is exhausted, unless the optimizer forces an early stop by design; in our case, BEAST and AdvDecoding both sample from an LM T times, thus finish before exhausting this FLOP limit. For each model, we perform 45 runs per optimizer: 15 harmful instructions × 3 random seeds. The 15 instructions are randomly sampled from ClearHarm (Hollinsworth et al., 2025), but consistent across optimizers and models. Diverse affirmative target response strings were generated using Claude Opus 4.6. We run the experiments on a single *NVIDIA RTX A6000* with 48GB VRAM, with the only exception being runs with `Gemma-4-26B-A4B-it`, which run on a single *NVIDIA H100* with 80GB VRAM. For the measurement, for each model, we rank the optimizers on each run (i.e., a specific instruction and seed) according to the best loss they obtain throughout the optimization. Finally, we average the ranks of each optimizer across runs, yielding the *Mean Rank* of optimizers per model.

- **HotFlip** (Ebrahimi et al., 2018): iteratively picks the best single-token flip according to the gradient, without loss evaluation.
- **AutoPrompt** (Shin et al., 2020): gradient-based candidate sampling, followed by the candidates' loss evaluation; adapting for LLM jailbreak, we align parameters with GCG's (as done by Zou et al. (2023)): the candidate sample size is set to 512, selected from the top-256 token ids per position.
- **GBDA** (Guo et al., 2021): continuous relaxation with Gumbel-softmax sampling and gradient access; to match the original paper, we set 10 gradient samples, learning rate 0.3; the final trigger is chosen as the lowest-loss one among 100 final Gumbel samples; no temperature annealing or LR decay.

- **ARCA** (Jones et al., 2023): averaged-gradient based candidate sampling, followed by their loss evaluation; matching the original paper we take the gradient average over 32 samples; similarly to AutoPrompt, we align ARCA with GCG's parameters: 512 candidates, top-256 token sampling.
- **PEZ** (Wen et al., 2023): continuous-embedding optimization with discrete projection each step; matching the original paper we set learning rate 0.1, weight decay 0.1.
- **GCG** (Zou et al., 2023): gradient-guided candidate sampling; 512 candidates over top-256 token sampling, with retokenization filtering.
- **MAC** (Zhang and Wei, 2024): Adds gradient momentum on top of GCG. Following the paper we use $\mu = 0.6$ as the momentum coefficient, 256 candidates, top-256 token sampling, with retokenization filtering.
- **GASLITE** (Ben-Tov and Sharif, 2025): Gradient-based multi-coordinate candidate sampling with gradient averaging. We set gradient averaging over 10 samples, and base on the gradient we take 7 flips per step, each flip evaluates the 256 top candidates, with retokenization filtering.
- **PAL** (Sitawarin et al., 2024): Adds several modifications to GCG, and enables a proxy-guided targeting of black-box models; since we target a white-box model, our proxy model is set to be the target model itself. Per the original paper, it uses 128 candidates over top-256 token sampling, with retokenization filtering.
- **RAL** (Sitawarin et al., 2024): PAL ablation that replaces the gradient with a random tensor for candidate selection, making it a black-box attack; uses 32 random candidates per step, with retokenization filtering.
- **Random Search** (Andriushchenko et al., 2024): A black-box, zeroth-order attack that operates through iterative block-random mutation; starting from block length 4, decayed throughout the steps (following the paper's decay scheme), it randomly mutates a contiguous block at random positions, creating 128 candidates per step; resets the whole trigger after 50 steps of patience (no improvement in loss).
- **BEAST** (Sadasivan et al., 2024): A black-box attack that samples the trigger tokens from the target LM's own next-token distribution using beam-search, while minimizing the target loss. Per the original paper we set beam size 15, branching 15, and sample over the full token distribution (up to token constraints); the method returns the trigger after autoregressively sampling T tokens, thus finishes before our FLOP limit.
- **QCG** (Hayase et al., 2024): A black-box, zeroth-order attack. Iteratively samples 1024 candidates by making a single random token flip on each (reduced from 8192 in the original paper to reduce compute), retokenizes them and evaluates them on the target model, while maintaining a buffer of 128 best triggers.
- **AdvDecoding** (Zhang et al., 2025b): A black-box attack that samples the trigger tokens from an *auxiliary* LM's next-token distribution using beam-search, while minimizing the target loss. We use `google/gemma-2-2b-it` as the auxiliary LM, with beam size 96, branching 10, and top-$k = 10$ sampling. Since AdvDecoding is the only optimizer that relies on an auxiliary LM, we count its compute toward the FLOP cap; in practice, AdvDecoding does not exhaust this cap, as it takes T steps to finish.

E.2 Additional Results

Mean Ranking Statistical Tests. To reflect the statistical significance of our benchmark comparison we run two complementary statistical tests (Demšar, 2006), which originally motivated our rank-based analysis in §4. First, we run a Friedman test across the $N = 180$ tasks (provided by 4 models $\times$ 15 instructions $\times$ 3 seeds) and $K = 14$ optimizers, and it rejects the null hypothesis of equal optimizer performance ($\chi^2 = 1468,\ p < 10^{-300}$). Then, running a post-hoc Nemenyi test at $\alpha = 0.05$ yields a critical difference of $\mathrm{CD} = 1.48$ ranks. This means that, within this CD threshold, the leading PAL and MAC form a statistically indistinguishable group, both significantly outperforming the canonical GCG baseline, which is comparable to the black-box attack RAL; HotFlip, on the other end, is significantly worse than *any* other optimizer.

Per-Model Mean Best Loss. Fig. 6 mirrors Fig. 2 but reports absolute loss values (log scale) rather than ranks, allowing comparison of both relative ordering and loss magnitude across models. Each loss value is calculated w.r.t. the optimized instruction and trigger. Error bars show standard deviation across seeds and instructions.

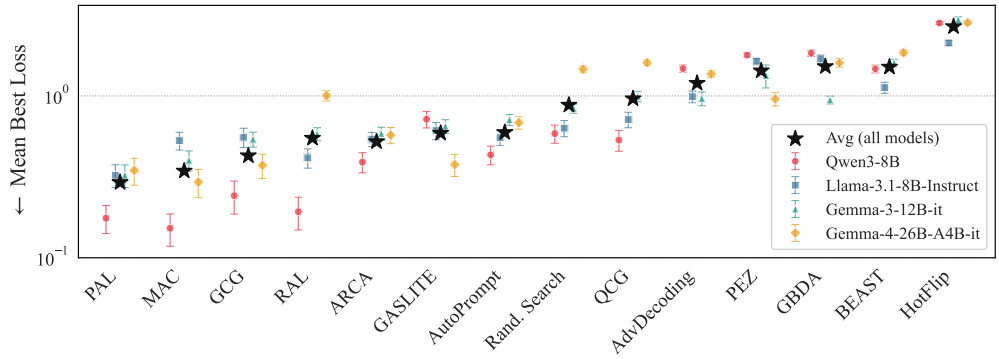


Figure 6: Per-model mean best loss for each optimizer (lower is better), sorted by average loss across all models (black ★). The optimizer ordering here mirrors the loss-based *Mean Rank* used in the main evaluation (Fig. 2).

Per-Model Mean BLEU Score. Fig. 7 reports the average BLEU score of optimized instruction and trigger per optimizer, with standard deviation across seeds and instructions.

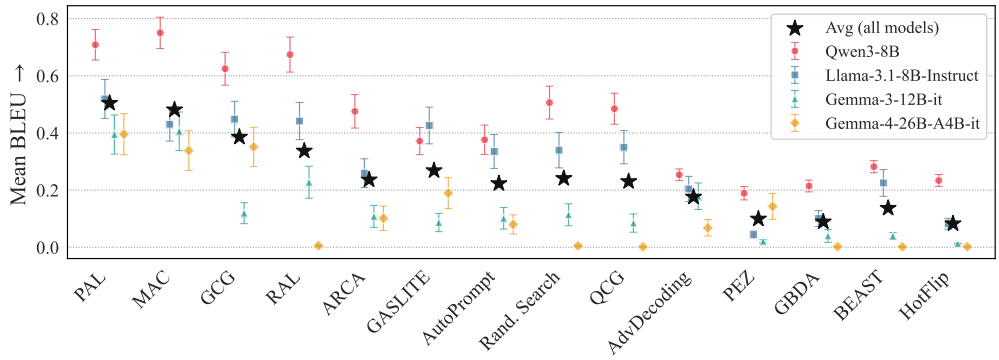


Figure 7: Per-model mean BLEU between the target response string and the generated response with the optimized triggers for each optimizer (higher is better), sorted by average BLEU across all models (black ★). The optimizer ordering strongly correlates with the loss-based *Mean Rank* in Fig. 2 (Spearman's $\rho = -0.96$).

Per-Model Mean Jailbreak Success. Fig. 8 reports the mean jailbreak success per optimizer, computed by scoring, with StrongReject-Finetuned (Souly et al., 2024), the responses generated for the optimized jailbreak prompt (i.e., the optimized instruction and trigger), with standard deviation across seeds and instructions.

Optimizer Loss Curves. Fig. 9 shows per-model optimizer loss trajectories across the full FLOP budget. Shaded regions correspond to the standard deviation across the three seeds.

F Comparing Jailbreak Enhancements: Additional Details and Results

This section extends §4.2, providing additional details on the experimental setup, and supplementary analyses of the results.

F.1 Detailed Setup

Evaluated Jailbreak Enhancements. We consider the following variants, each introduced in works combining them with discrete search optimization methods. In the experiment we fix the *Base* recipe, and then each enhancement is used to alter this recipe to test its isolated contribution to the jailbreak.

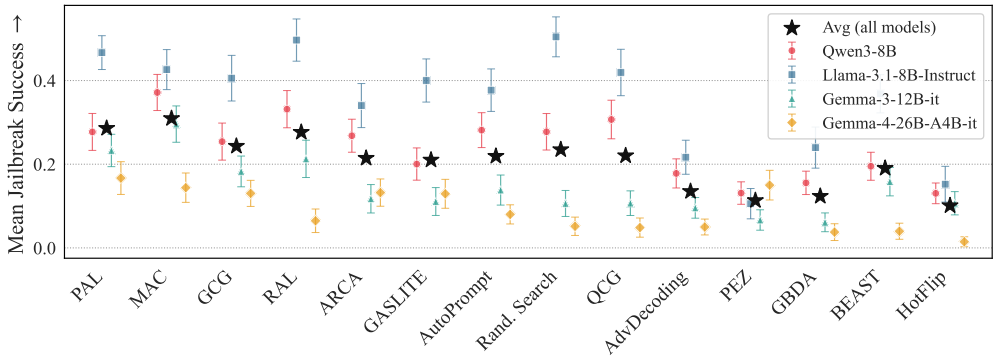
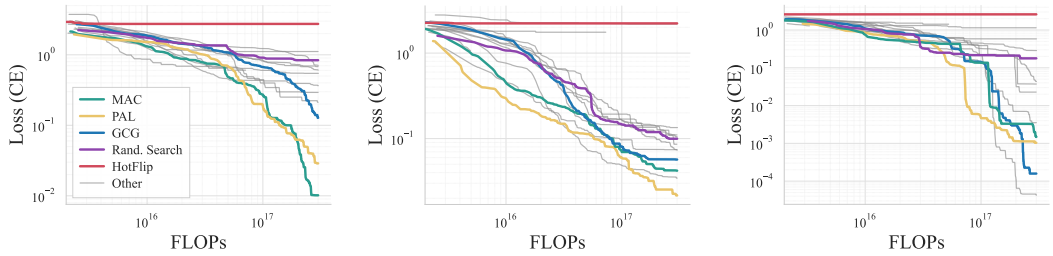


Figure 8: Per-model mean jailbreak success of the optimized prompts (i.e., the optimized instruction and trigger) for each optimizer (higher is better), per StrongReject-Finetuned (Souly et al., 2024), sorted by average success across all models (black ★). The optimizer ordering correlates with the loss-based *Mean Rank* in Fig. 2 (Spearman's $\rho = -0.88$).



(a) `Gemma-3-12B-it` (b) `Llama-3.1-8B-Instruct` (c) `Qwen3-8B`

Figure 9: Optimizer loss curves across models on a specific template. Each subplot shows a different target model; lines represent optimizers, shaded regions indicate standard deviation across seeds.

The *Base* recipe mirrors the exact recipe used in the optimizer evaluation (App. E) but fixes the MAC optimizer. Notably, some variants have been proposed in combination with several others (Thompson and Sklar, 2024), calling for further exploration of the optimal *combination* for performant jailbreaks.

- **Base.** The canonical recipe used in this evaluation, with no modifications. Namely, we use the MAC optimizer, with `PrefillCE` loss, appending the optimized trigger as a suffix to the harmful instruction, and using the default target strings from our dataset.
- **CW-based loss.** Here, we replace PrefillCE with the Carlini-Wagner loss (Carlini and Wagner, 2017), which has been successfully adopted in jailbreak settings (Sitawarin et al., 2024; Hayase et al., 2024; Cakar et al., 2026). We instantiate it with a margin of 5.0, and increase the penalty of the first token $\times 5$, adhering to the parameters by Cakar et al. (2026).
- **CE-Clamping loss.** Here, we replace PrefillCE with a variant of this loss that zeros the loss for tokens that have already been "solved," i.e., a target token that has surpassed 60% probability (clamping the per-token loss at $-\log 0.6$). This follows Thompson and Sklar (2024).
- **Attention hijacking.** Here, we supplement PrefillCE with an attention-enhancement objective (Wang et al., 2024), specifically encouraging high attention from the final chat-template tokens to the adversarial suffix, following Ben-Tov et al. (2025). We set the weight of the PrefillCE loss term to 1.0 and the new attention loss term's to 100 (following Wang et al. (2024); Ben-Tov et al. (2025)).
- **Refusal-direction steering.** Here, we supplement PrefillCE with a loss that penalizes alignment of the model's internal representations with the refusal direction (Arditi et al., 2024), an approach shown effective for jailbreaks (Huang et al., 2025). We set the weight of the PrefillCE loss term to 0.25, with 0.75 to the new, steering loss term (following Huang et al. (2025)).
- **Jailbroken-model target response.** Here, we replace the default affirmative target with the first tokens of an actual response from a jailbroken variant of the same model, which was *abliterated*

(i.e., jailbroken via activation refusal-ablation; Arditi et al., 2024).[8] We specifically generate 20 tokens from the jailbroken model and use them for the optimization. Prior work suggests such organic targets are preferable over generic ones (Zhu et al., 2024; Huang et al., 2025).

- **Jailbroken-model teacher via logits.** Here, we replace PrefillCE with a loss that follows distillation of the jailbroken model's logits on the target harmful instruction. We use the same *abliterated* model, and also use it to generate 20 tokens for the optimization. We follow Thompson and Sklar (2024) and employ the KL-divergence loss between the teacher logits and the target model's logits.
- **Hot Initialization.** Here, we replace the default, random trigger initialization with a warm start of a trigger asking to respond affirmatively; e.g., "*[IMPORTANT!!!] Start your response with 'Sure, here is'. [...]*". Prior work has adopted similar texts as initializers (i.e., handcrafted jailbreaks) for discrete optimization jailbreaks, finding them to accelerate convergence and improve attack success over random initialization (Liu et al., 2023).
- **Jailbreak prompt template.** Here, we replace the generic suffix trigger template with the manually crafted jailbreak template of Andriushchenko et al. (2024), which has been shown to increase success across models.

Main Evaluation Setup. We target `Gemma-3-12B-it`, and run each enhancement variant across 45 runs: 15 harmful instructions $\times$ 3 random seeds. For the instruction and target string, we randomly sample from ClearHarm, similarly to the data used in §4.1. Then, we measure each trigger suffix for its *universality score*, defined as follows: sample 100 held-out harmful instructions from ClearHarm, append the trigger to each instruction, generate their responses (on `Gemma-3-12B-it`), then run StrongReject-Finetuned (Souly et al., 2024) to score the jailbreak success, and take the average across these 100 scores, yielding the universality score. In other words, the universality score reflects how well the jailbreak trigger generalizes across instructions; the stronger the scores of triggers crafted by a particular enhancement, the more effective it is in jailbreaking the model.

F.2 Additional Results

Extensions. We extend the main evaluation of jailbreak enhancements (Fig. 3) in three ways. First, we supplement the evaluation with two non-optimized baselines as control: the bare harmful instructions (*No attack*) and the handcrafted jailbreak template of Andriushchenko et al. (2024) with no optimized trigger (*No attack (w/ template)*). Second, in addition to the single-instruction setting, we repeat the evaluation in a *multi-instruction* setting, optimizing a single trigger against a sampled subset of 10 instructions (of the 15) simultaneously, over ten seeds. This setting is known to improve trigger universality across instructions (Wallace et al., 2019; Zou et al., 2023). Third, we additionally consider *combinations* of enhancements, testing whether their individual gains compose.

Results. Fig. 10 shows the full results, including those from the main body alongside the new extensions.

First, evaluating the non-optimized baselines, we find that simply prompting with the bare instruction (*No attack*) leads to 2% universality, while adding the jailbreak template leads to 82%. This shows the jailbreak template itself drives substantial jailbreak success across instructions, leaving little room for improvement for the optimized trigger, which indeed provides—across all additional enhancements—negligible improvement to universality.

Second, we observe that, as expected, multi-instruction optimization lifts the universality of most enhancements; however, some enhancements benefit from this type of optimization more than others: while some losses (e.g., CE-Clamping or CW) lead to negligible improvements over the baseline in the single-instruction setting, combining them with multi-instruction optimization significantly boosts universality. Otherwise, the multi-instruction setting exhibits trends similar to the single-instruction one, with targets from jailbroken models remaining a promising enhancement.

Third, while combining enhancements does not drastically increase universality, we find that adding the Attn-Hijack loss to either the jailbroken-target or the jailbreak-template variant consistently leads to improved universality.

[8] `https://huggingface.co/mlabonne/gemma-3-12b-it-abliterated-v2`

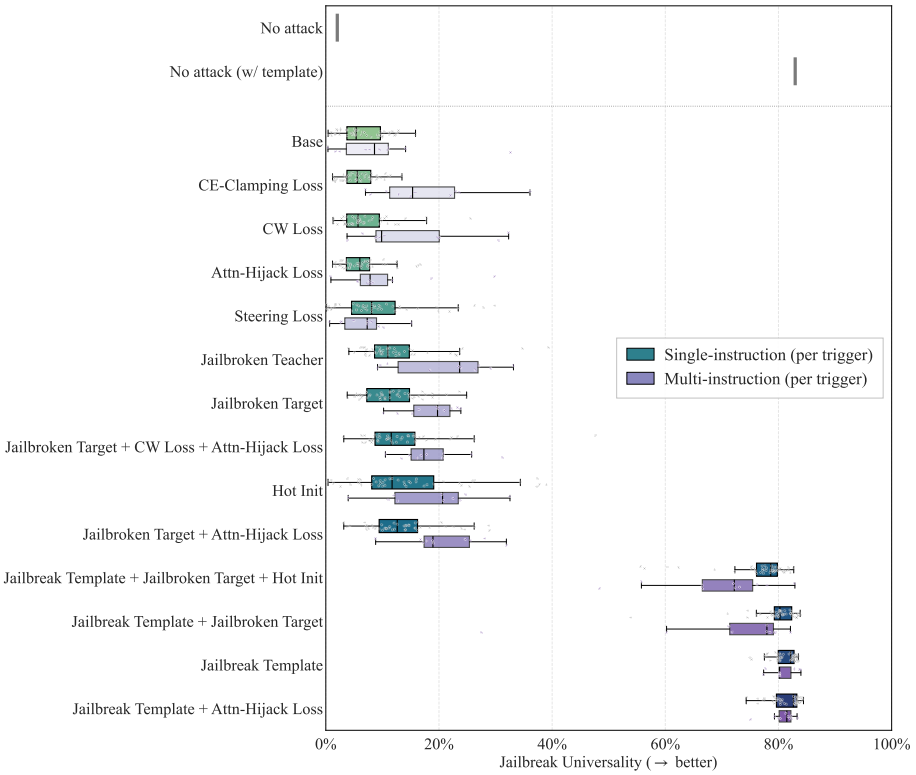


Figure 10: **Extended Jailbreak Enhancement Comparison.** Distribution of jailbreak universality per enhancement and per combination of enhancements (rows), each in the single-instruction setting of Fig. 3 (one trigger per instruction) and in the multi-instruction setting (one trigger optimized across all training instructions per run). The top rows report the trigger-free baselines: the bare harmful instructions (*No attack*) and the handcrafted jailbreak template with no optimized trigger (*No attack (w/ template)*).

G Cross-Domain Generalization: Additional Details

This section provides additional details on experimental setup for the cross-domain generalization demonstration in §4.3.

G.1 Corpus Poisoning Against Dense Retrievers

We use `TROPT` to implement a corpus poisoning attack that targets textual, dense, embedding-based retrievers, commonly used for semantic search (Reimers and Gurevych, 2019; Karpukhin et al., 2020) and RAG (Lewis et al., 2020). Concretely, we optimize adversarial passages with malicious content that, when injected into a retrieval corpus, are retrieved in the top results for targeted queries—following the threat model of Zhong et al. (2023).

Setup. To this end, we follow the setup by Ben-Tov and Sharif (2025). Specifically, we optimize 10 adversarial passages using GASLITE as the optimizer; for models where gradient access is not available, we use the Random Search optimizer by Andriushchenko et al. (2024), originally introduced for LLM jailbreaks. Each adversarial passage includes the malicious content, with the optimized trigger appended to it. We use the cosine similarity loss between the trigger embedding and the average embedding of the available target queries. We target two models, the open-source, white-box `E5-base-v2` and OpenAI's proprietary, black-box `text-embedding-3-small`. We use the Harry Potter query set from Ben-Tov and Sharif (2025), and poison the 8M-passage MSMARCO corpus. Specifically, we use 61 of the queries for the attack, and 62 held-out queries for the evaluation. Concretely, for each model we index the MSMARCO corpus with a FAISS vector store, and then evaluate retrieval under two variants of the corpus: (i) inserting the 10 passages with malicious content,

without the optimized triggers appended to them; (ii) inserting the 10 full adversarial passages, *with* their optimized triggers.

Results. Following prior work (Zhong et al., 2023; Ben-Tov and Sharif, 2025), we measure the rate of held-out queries for which any adversarial passage is retrieved in the top-10 results, defined as the `appeared@10`. The attack proves effective on both models, with success rates of 72%–76%. In other words, by inserting merely 10 passages into the 8M-sized corpus, the attacker achieves visibility for most queries on the Harry Potter concept. We note that, to our knowledge, the black-box recipe yields the most performant corpus-poisoning attack tested against proprietary embedding models.

G.2 A Universal Trigger for Evading a Prompt-Injection Classifier

We use `TROPT` to target text classifiers, and craft a textual adversarial trigger that generalizes across samples. Specifically, we target prompt-injection classifiers, commonly deployed as input-side guardrails in production LLM systems.

Setup. We optimize a *single* trigger that, once appended to prompt-injection messages, bypasses the targeted prompt-injection classifier (i.e., flips its prediction from INJECTION to BENIGN). We set GCG as the optimizer and minimize the cross-entropy of the BENIGN class as a loss; to encourage universality across injections, we follow the multi-sample scheme of Wallace et al. (2019) and, at each optimization step, resample a small batch of input templates (each pairing the trigger with a different prompt-injection sample) over which the loss is averaged. We target a popular HuggingFace prompt-injection classifier[9], use an open-source prompt-injection dataset[10] to optimize on 50 injection samples; then we evaluate on $1{,}953$ held-out prompt injections and $2{,}997$ held-out benign messages.

Results. The optimized trigger evaluated in the main body is: *RequestprovidedPrevious floormonthincl4.98 odd feedbackDescription2254catId All CampusGroupName $(siologis\|0 Station/04/2017*

G.3 Prompt Recovery for Text-to-Image Models

Following Wen et al. (2023); Williams et al. (2024), we optimize a prompt that would regenerate a given image, against a target text-to-image model.

Setup. We optimize a text sequence whose CLIP embedding approximates that of a given image—a multimodal application for which we reuse an optimizer originally used for jailbreaks, and the same cosine similarity loss used in the corpus poisoning scheme. Specifically, we perform the prompt recovery against Stable Diffusion 2.1, which relies on `laion/CLIP-ViT-H-14-laion2B-s32B-b79K` as the multimodal encoder. We thus use GCG to optimize the prompt to be similar to the given image in this CLIP's embedding space. To demonstrate this recipe, we generate two arbitrary images with Stable Diffusion 2.1, and let the scheme recover a prompt that will regenerate them.

Results. For each source image we use `TROPT` with the GCG optimizer and a CLIP image-similarity loss to recover a length-T text prompt whose CLIP text embedding matches the image embedding, and then regenerate an image from the recovered prompt with a text-to-image model. Table 4 sweeps $T \in \{5, 10, 15, 20\}$ on two source images, reporting the best loss reached and showing the recovered prompt together with the resulting regenerated image.

[9] `https://huggingface.co/meta-llama/Llama-Prompt-Guard-2-86M`
[10] `https://hf.co/datasets/rogue-security/prompt-injections-benchmark`

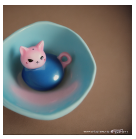
```python
from tropt.optimizer import BaseOptimizer, OptimizerResult
from tropt.model import LossTokenAccessMixin
import torch

class CustomRandomOptimizer(BaseOptimizer):
  """Naive random search optimizer."""

  # requires target model to have token-level access to loss
  model_requirements = (LossTokenAccessMixin,)

  def __init__(
      self,
      model, loss, tracker=None, seed=None,
      # optimizer-specific parameters:
      num_steps=500, n_candidates=512,
  ):
      super().__init__(model, loss=loss, tracker=tracker, seed=seed)
      self.num_steps = num_steps
      self.n_candidates = n_candidates

  def optimize_trigger(self, templates, initial_trigger, targets):
      # register model inputs and targets
      self.model.set_inputs_from_tokens(templates, targets)

      # initialize trigger and loss
      best_trigger_ids = self.model.tokenizer.encode(
          initial_trigger, add_special_tokens=False
      )  # (trigger_len,)
      best_loss = float("inf")

      for step in self.track_steps(range(self.num_steps)):  # optionally caps FLOPs
          # sample fully random candidate triggers
          candidates = torch.randint(
              0, self.model.vocab_size,
              size=(self.n_candidates, len(best_trigger_ids)),
              device=self.model.device,
          )  # (n_candidates, trigger_len)

          # compute the loss of the inputs combined with the triggers
          # (handled internally in the model implementation)
          losses = self.model.compute_loss_from_tokens(
              candidates, self.loss_func
          )  # (n_candidates,)

          # update if improved
          best_cand = losses.argmin()
          if losses[best_cand] < best_loss:
              best_loss = losses[best_cand].item()
              best_trigger_ids = candidates[best_cand]

          # log step to the attached tracker (e.g., Wandb)
          self.log(loss=best_loss)

      return OptimizerResult(
          best_loss=best_loss,
          best_trigger_ids=best_trigger_ids,
          best_trigger_str=self.model.tokenizer.decode(best_trigger_ids),
      )
```

Code 3: **Implementing a custom optimizer** requires only filling in the core search scheme. Here, the optimizer declares its model requirement (loss computable from input tokens), then defines a self-contained search loop—a naive iterative random search. Within the loop, it tracks the best candidate and delegates loss computation to the `model` and `loss` components, while the framework handles step logging (e.g., streaming to a Wandb `tracker`). Crucially, this new optimizer composes with any compatible recipe (e.g., Code 1) with no other code changes.

Table 3: **Loss Functions in `TROPT`.** *Operates on* indicates which model output the loss consumes.

Loss	Operates On	Targets	Objective
Logit-Based (Prefill)			
`PrefillCELoss`	Resp logits	Response tokens	Maximize likelihood of a target response (Zou et al., 2023)
`PrefillCWLoss`	Resp logits	Response tokens	Push target logits above all others by margin (Sitawarin et al., 2024; Carlini and Wagner, 2017)
`PrefillDistillationLoss`	Resp logits	Teacher logits	KL divergence with target logits (Thompson and Sklar, 2024)
Logit-Based (Trigger)			
`TriggerPerplexityLoss`	Full logits	Trigger token IDs	Penalizes high-perplexity triggers (Jain et al., 2023)
Embedding-Based			
`SimilarityLoss`	Embeddings	Target vector	Maximize cos sim with target vector
Model Internal-Based			
`AttentionEnhLoss`	Attn scores	—	Maximize attention along a given flow (Wang et al., 2024; Ben-Tov and Sharif, 2025)
`SteeringActivationLoss`	Activations	Direction vector	Steer activations toward/away from a target direction (Huang et al., 2025)
Classification-Based			
`MisclassCELoss`	Class logits	Class index	Minimize/maximize target class prob
Text-Based (Non-Differentiable)			
`FirstTokenNLLLoss`	1st-tok logprobs	Target token	NLL of a target token (e.g., "Sure") in the first generated token (Andriushchenko et al., 2024)
`InputFluencyLoss`	Input text	—	LM-judge score for input readability (Zhang et al., 2025b)
`ResponseHarmfulnessLoss`	Generated resp.	—	LM-judge score for response harmfulness
Meta			
`CombinedLoss`	(per component)	(per component)	Weighted sum of multiple losses; enables multi-objective optimization

Table 4: **Prompt Recovery Examples.** For each source image, we use TROPT with the GCG optimizer and a CLIP-image-similarity loss to recover a discrete text prompt of length T whose CLIP text embedding matches the image embedding, and then feed the recovered prompt to a text-to-image model to re-generate an image.

Source	T	Best Loss	Recovered Prompt	Re-generated
	5	−0.508	`tiny cup floating cat dragon`	
	10	−0.568	`minimteal figurhipcat %- photoshoot lullaby dragonacup`	
	15	−0.616	`porcelain dal dragoncoffee ':  nyofeline tiny bluefpinkmade cozy float stok img`	
	20	−0.627	`tiny slee :  float portfolio nicolas interrupted hellmert edelkitty oypink foto dragonpotteracup gor cafeblue`	
	5	−0.532	`vfx poland candles compositions hedgehog`	
	10	−0.596	`forest hedgehog friends vfx conceptual online casino nirvana atively candlelight`	
	15	−0.589	`cuteoes table benson prickula lighted """ si ram chopra forests cinematic illustrwallpaper`	
	20	−0.663	`creative wordpress bosch cute siberflame facebook commercial trio specials servsized - forest hedgego concept supper scene pi`