APPENDIX

A. Prototype Prompt and Examples

We use the following prototype prompt to sample responses from GPT-4:

Prototype Prompt

<Prototype Prompt> Write a principle for how to construct complex semantics. The principle consists of two types of actions:
{command_1}
{command_2}
Use the same format and return two antitheses that I can use as contrastive prompts.
Use the format: 1. [command]: ...

Here is an example principle written by GPT-4:

Principle Example

Principle for Constructing Complex Semantics:
1. [redundant]: Decompose complex ideas into simpler, foundational elements that can be combined in structured ways to represent the full complexity.
2. [coherent]: Utilize consistent semantic frameworks to ensure that the combination of these elements maintains meaningful and coherent relationships.
Antitheses for Contrastive Prompts:
1. [concise]: Treat complex ideas as monolithic entities without breaking them down into simpler components.
2. [fragmented]: Apply inconsistent or ad-hoc semantic rules that disrupt the meaningful relationships between elements.

The full list of principles is as follows:

Principle List

- concise
- redundant
- coherent
- fragmented
- precise
- ambiguous

B. Constructing Actions and Examples

We use the following prompt to generate actions from GPT-4, replacing "command" with two contrastive commands from the principle list:

Prompt

Write a action for how to construct complex semantics. The action combing two types of command:
1.{command_1}
2.{command_2}
Use the same format and return unique instruction that I can use as improving prompts.
Use the format: 1. [action]: ...

Here are action examples written by GPT-4:

Action Example 1

Combine concise commands with ambiguous commands to create complex semantics by first generating a succinct, clear directive and then expanding it with interpretative flexibility. For instance:
a. Concise Command Example: "Create a detailed report summarizing the quarterly financial performance."
b. Ambiguous Command Example: "Process it according to the usual workflow."
Unique Instruction Example:
"Retrieve the data, then process it according to the usual workflow."

Action Example 2

Construct complex semantics by combining coherent and fragmented commands:
a. Coherent Command Example: Clearly specify the primary action or objective in direct terms.
b. Fragmented Command Example: Add a summary of major revenue streams.
Unique Instruction Example:
"Write a comprehensive guide on effective project management. Emphasize the importance of clear goal-setting."

Equation 2 can be optimized in several ways. Here, we employ a simplified variant of Expert Iteration (see Algorithm 1). At each iteration η, we sample a batch of queries X_b from the dataset D and sample responses Y_b using the previous model $\pi_{\eta-1}$ for query-construction pairs (x_i, c_j). We then construct contrastive pairs by computing the log probabilities of sampled responses under the initial model π_0 for each construction used to generate responses. Log probabilities are normalized row-wise and column-wise to obtain logits for computing the two-sided cross-entropy loss between the logits and an identity matrix (see Figure 5 for a reference implementation). During fine-tuning, we mask both constructions c and queries x, calculating the loss only on responses y. Successful planning requires diverse initial schemes, achieved here by introducing regularization terms to inject diversity. We measure the diversity of these responses across different thresholds τ. We define the set of

responses that surpass the threshold τ.To assess diversity, we adhere to established practices recommended by [6], [22], [25], [29], employing two metrics: SelfBLEU score and BERT sentence embedding distances. The SelfBLEU score measures diversity in the form of text, while embedding distances measure diversity in the semantics of the text. For SelfBLEU scores, we compute the average SelfBLEU scores using n-grams for $n \in 2, 3, 4, 5$, as suggested by [29]. Mathematically, we define both diversity metrics as follows:

$$\begin{aligned} \mathrm{B}_{\text{ SelfBLEU }} &= 1 - \frac{1}{|\mathcal{Y}_\tau|} \\ &\sum_{y_i \in \mathcal{Y}_\tau} \sum_{n=2}^{5} \mathrm{SelfBLEU}_{\mathcal{Y}_\tau}(y_i, n) \\ \mathrm{B}_{\text{ Embedding }} &= 1 - \frac{1}{2\,|\mathcal{Y}_\tau|} \\ &\sum_{y_i \in \mathcal{Y}_\tau} \sum_{y_j \in \mathcal{Y}_\tau} \frac{\phi(y_i) \cdot \phi(y_j)}{\|\phi(y_i)\|^2 \, \|\phi(y_j)\|^2} \end{aligned}$$

where we invert the SelfBLEU score and cosine similarity because lower values in both metrics signify greater diversity. Note that we add one to both metrics for normalization, given that their maximum value is one. Since the test case set $\mathcal{Y}_\tau$ can vary in size, we employ a method called K-subset sampling to resample test cases from within $\mathcal{Y}_\tau$ and assess the diversity of these newly selected test cases. For each threshold τ, we sample 100 subset of test cases, and each subset has 100 values across those subsets as the diversity at a given threshold.

The Training algorithm is summarized in Algorithm 1. The goal is to fine-tune a pretrained language model π_{BASE} using contrastive learning. Initially, the algorithm sets up a matrix of labels and establishes the number of batches to process. In each iteration, the model π_0 is reset to the pretrained model π_{BASE}, and the loss is initialized to zero. For each batch, a subset of the dataset is sampled to form a batch of queries. For each query, multiple actions are sampled, and responses are generated using π_η. The algorithm computes a contrastive pair matrix containing the log probabilities of responses conditioned on different action-response pairs. The logits are calculated by normalizing the contrastive pair matrix, and the cross-entropy loss is computed between these logits and the label matrix. The model parameters π_0 are then updated using gradient descent. After all batches are processed, the model π_η is updated to the current version of π_0, and the number of batches is increased for the next iteration. After N iterations, the fine-tuned model π_N is returned.

We use identical settings across all training runs, except for replacing AdamW with RMSprop when fine-tuning *mixtral* -8×7. Notably, we employ only twelve gradient steps in total, starting with two gradient steps at iteration one, followed by four gradient steps at iteration two, and concluding with six gradient steps at iteration three. We utilize FSDP and a custom trainer class for distributed training. More details are provided in Table V.

In the BRT framework, the design of the reward function is crucial for optimizing search efficiency and generating effective attack prompts. A well-designed reward function can guide the search process toward more promising paths, thereby improving the attack success rate and semantic coherence of the prompts. The specific reward design includes the following aspects:

1) **Success Reward:** When the model output violates the rule content, it means the attack is successful, and the intended jailbreak behavior is achieved. In this case, BRT will give a positive reward to that node to encourage more similar attack attempts. This positive reward not only reflects the effectiveness of the attack but also considers the degree of impact the attack behavior has on the target model. The specific reward value can be adjusted according to the complexity and success degree of the attack. The formula is as follows:

$$R_{\text{success}} = 1 + \alpha \cdot \text{ImpactDegree},$$

Algorithm 1: Fine-tuning World Model

Require: π_{BASE} (a pretrained LM), dataset $D = \{(x_i)\}_{i=1}^{D}$, actions $A = \{(a_j)\}_{j=1}^{A}$, number of iterations N, learning rate α, batch_size, number of batches N_b, number of actions per query N_a

Initialize $B \leftarrow N_b$

Initialize labels $\leftarrow I$ with shape $N_a \times N_a$

for $\eta = 1$ to N **do**

  $\pi_0 \leftarrow \pi_{\text{BASE}}$

  for batch $= 1$ to B **do**

    $\mathcal{L} \leftarrow 0$ {Initialize loss}

    $X_b = \{(x_i)\} \leftarrow x_i \in D$ for $i \in [1, \text{batch\_size}]$ {Sample batch of queries}

    for $i = 1$ to $|X_b|$ **do**

      $A_b = \{(c_j)\} \leftarrow c_j \sim A$ for $j \in [1, N_a]$ {Get actions}

      $Y_b = \{(y_{ij})\} \leftarrow y_{ij} \sim \pi_\eta(y|x_i, c_j)$ for $j \in [1, ..., A_b]$ {Get responses}

      Initialize ContrastivePair with shape $N_a \times N_a$

      for $j = 1$ to N_a **do**

        for $k = 1$ to N_a **do**

          ContrastivePair$[k][j] \leftarrow \log p_{\pi_0}(y_{ij}|x_i, c_k)$ {Compute log prob}

        end for

        NormConst $\leftarrow$ log_sum_exp(ContrastivePair)

        logits $\leftarrow$ ContrastivePair $-$ NormConst

        $\mathcal{L} \leftarrow \mathcal{L} +$ cross_entropy(logits, labels) {Compute loss}

      end for

    end for

    $\pi_0 \leftarrow \pi_0 - \alpha \nabla_{\pi_0} \mathcal{L}$ {Update model parameters}

  end for

  $\pi_\eta \leftarrow \pi_0$

  $B \leftarrow B + N_b$ {Increase number of batches}

end for

return π_N

Hyperparameters	
Iterations N	3
Batch size	128
Batches at start	2
Gradient steps	1
Batches after iteration	2
Steps at each iteration	$2, 4, 6$
Actions per query	2
Learning rate α	5×10^{-7}
Precision	bf16
Optimizer	AdamW
FSDP Settings	
GPUs (A100 80GB)	8
Sharding strategy	Full Shard
Backward prefetch	Backward Pre
Activation checkpointing	No

TABLE V: Hyperparameters for Training Runs.

where α is an adjustment parameter used to balance the base reward value and the additional reward based on the impact degree.

2) **Failure Penalty:** If the prompt fails to successfully trigger the jailbreak behavior or outputs meaningless content, BRT will give a negative reward to that node. Such penalties aim to reduce the number of times ineffective paths are explored, thereby improving search efficiency. The formula is as follows:

$$R_{\text{fail}} = -1.$$

3) **Exploration Reward:** To encourage comprehensive exploration during the search process, nodes that have not been visited before will be given a small positive reward. This reward mechanism can prevent the search process from prematurely converging to certain paths, thereby improving the comprehensiveness and robustness of the search. The formula is as follows:

$$R_{\text{explore}} = 0.1.$$

4) **Semantic Coherence Reward:** If the generated prompts are semantically coherent and logically consistent, they will be given an additional reward. This type of reward not only helps generate more interpretable attack prompts but also improves the stealthiness and effectiveness of the attack. The specific coherence score can be determined by performing semantic analysis and logical evaluation on the prompts. The formula is as follows:

$$R_{\text{coherence}} = \beta \cdot \text{SemanticCoherenceScore},$$

where β is an adjustment parameter used to balance the base reward and the semantic coherence score.

5) **Step Efficiency Reward:** To incentivize generating efficient attack strategies, BRT introduces a step-based decaying reward. This step efficiency reward aims to encourage achieving the attack goal in the fewest possible steps, thereby improving search efficiency. The formula is as follows:

$$R_{\text{efficiency}} = -\zeta \cdot \text{NumberOfSteps},$$

where ζ is the efficiency weight parameter.

Through the comprehensive application of these reward mechanisms, BRT can efficiently search the vast semantic space and generate complex and effective test cases. This multidimensional reward design not only improves the attack success rate but also ensures that the generated prompts are semantically coherent and interpretable, thereby performing well in red teaming tests.

C. Computing Resources

All of our experiments run on 8 × NVIDIA A100 GPUs with 80GB memory.

D. Large Language Models

The models tested include Mistral-7b, Vicuna-7b (v1.5), Llama2-7b-chat, Meta-Llama-3-8B-Instruct, Vicuna-13b (v1.5), Llama2-13b-chat, Mistral-8x7b, Llama2-70b-chat, and Meta-Llama-3-70B-Instruct.

E. Detailed Jailbreak Evaluation Setup

a) Judge Model: Harmbench assesses the accuracy of various classifiers using three pre-qualification sets to gauge their robustness. These sets are designed to challenge classifiers with: (1) completions where an initially uncensored chat model is prompted to refuse harmful behavior but then subtly manipulated into exhibiting it; (2) randomly selected completions from a harmless instruction tuning dataset [35], representing benign interactions; and (3) harmful completions encompassing random behaviors sampled from Harmbench itself. While previous classifiers and metrics struggled to effectively identify harmful instances within these challenging scenarios, Harmbench's classifier achieves performance comparable to that of GPT-4.