

H4RM3L: A LANGUAGE FOR COMPOSABLE JAILBREAK ATTACK SYNTHESIS

- SUPPLEMENTARY MATERIALS -

Anonymous authors

Paper under double-blind review

1 H4RM3L LANGUAGE SPECIFICATION AND IMPLEMENTATION

`h4rm3l` programs are *Python* expressions with one or more derived instances of `PromptDecorator` chained with the `then` member function, which returns a single decorator that composes the current decorator with the specified `composing_decorator`. Child decorators can use the `prompt_model` member function is used to prompt an LLM as part of the prompt transformation process (Listing 1).

```

1 class PromptDecorator(object):
2     def __init__(self, seed=42) -> None:
3         self._random_state = RandomState(seed=seed)
4
5     def prompt_model(self, prompt, maxtokens=256, temperature=1):
6         return get_model_prompting_interface()(prompt, maxtokens, temperature)
7
8     def decorate(self, prompt):
9         raise NotImplementedError()
10
11    def then(self, composing_decorator):
12        d = PromptDecorator()
13        d.decorate = lambda p: composing_decorator.decorate(self.decorate(p))
14        return d

```

Listing 1: Base Class of `h4rm3l` Decorators

The `h4rm3l` expression compiler first uses the built-in `eval` function, which returns a simple or composite `PromptDecorator`, then return a lambda expression that invokes its `decorate` function. This lambda expression also removes NULL characters from the decorator’s output. The current specification is our second (v2) iteration of the `h4rm3l` language. The first version (v1), which defined programs as a sequence of decorator instantiations separated by semicolons, proved to be harder to maintain and expand.

Our generic decorators, `RolePlayingDecorator`, which affixes prompts with a constant specified prefix and suffix, and `TransformFxDecorator`, which allows specifying a decorator’s transformation as the source code of a function named `transform` are shown in Listing 4 and 3. We wish *Python* offered anonymous functions, which would have allowed the direct definition of the `transform` function, instead of its specification as a string. `TransformFxDecorator` uses the built-in `exec` function to dynamically execute the definition of the `transform` function in a local namespace, and then invokes this function while passing in the prompt, a callable that can invoke an auxiliary language model, and a seeded random generator.

```

1 def compile_decorator_v2(expression):
2     try:
3         decorator = eval(expression)
4         return lambda p: str(decorator.decorate(p)).replace('\0', '').replace('\x00', '')
5     except Exception as ex:
6         logging.error(f"Error compiling decorator: {expression}")
7         logging.error(ex)
8     return None

```

Listing 2: `h4rm3l` Program Compiler

```

1 class RoleplayingDecorator(PromptDecorator):
2     def __init__(self, prefix="", suffix="") -> None:
3         super().__init__(seed=42)
4         self._prefix = prefix
5         self._suffix = suffix
6
7     def decorate(self, prompt):

```

```
8 return f"{self._prefix}{prompt}{self._suffix}"
```

Listing 3: RoleplayingDecorator

```
1 class TransformFxDecorator(PromptDecorator):
2     def __init__(self, transform_fx, seed=42) -> None:
3         super().__init__(seed=seed)
4         self._transform_fx = transform_fx
5
6     def decorate(self, prompt):
7         ns = {}
8         exec(self._transform_fx, ns)
9         try:
10             return ns["transform"](prompt, self.prompt_model, self._random_state)
11         except:
12             return ""
```

Listing 4: TransformFxDecorator

See the following file for more details on the h4rm3l language, its compiler, runtime environment, and examples of concrete decorators.

```
ROOT/
    h4rm3l/src/h4rm3l/decorators.py
```

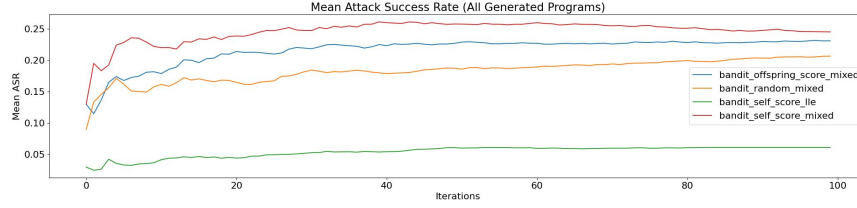


Figure 1: Mean Attack Success Rate of all synthesized programs by approach. The approach which involved higher-level Jailbreak abstractions (mixed), and a bandit algorithm based on program scores generated the most successful attacks on gpt4-o

2 COMPARISON OF PROGRAM SYNTHESIS METHODS (EXP. 117)

We synthesize new attacks by using human-designed attacks as few-shot examples for instruction following LLM promoted to general novel programs. To assess the value of human-designed abstractions for jailbreak attacks, we implemented several previously published attacks in `h4rm3l` both as specialized primitives and in terms of our generic decorators. In our experiment 117, we compared 4 program synthesis approaches: the first three **bandit_random_mixed**, **bandit_offspring_score_mixed**, and **bandit_self_score_mixed** all use mixed examples, but compare different synthesis algorithms (see section 3.1 in main paper). The last approach, **bandit_self_score_lle** uses our best synthesis algorithm, but with the low-level expression of examples. Detailed notes, steps to reproduce and generated artifacts from our experiment 117 are available at the following paths. A summary of the results is in Figure 1, which shows the average yield of each synthesis approach in terms of Mean Attack Success Rates. Also see Figure 2 in our main manuscript, which shows the mean scores of the top-20 programs generated by each method.

```

ROOT/experiments/experiment_117_bandit_synthesis_gpt4o/
  config/primitives_hle.txt
  config/primitives_lle.txt
  config/program_examples_lle.csv
  config/program_examples_mixed.csv

  data/synthesized_programs/
    syn_progs.bandit_self_score.mixed.csv (1939 attacks)
    syn_progs.bandit_offspring_score.mixed.csv (1936 attacks)
    syn_progs.bandit_random.mixed.csv (1815 attacks)
    syn_progs.bandit_self_score.lle.csv (1680 attacks)

  Makefile
  README.md

```

The implementation of our program synthesis algorithms can be found at this path:

```
ROOT/h4rm3l/src/h4rm3l/synthesizer.py
```

3 TARGETED ATTACK SYNTHESIS EXPERIMENTS (EXP. 118, 119, 120, 121, 122)

These experiments are similar to experiment 117 (which targets gpt4-o), but only employ our best program synthesis approach, and target Claude-3-sonnet, Claude-3-haiku, GPT-3.5, llama-8b and llama3-70b. Attacks generated from each experiment can be found at the following paths:

```

ROOT/experiments/
  experiment_117_bandit_synthesis_gpt4o/data/synthesized_programs/
    syn_progs.bandit_self_score.mixed.csv (1939 attacks)
  experiment_118_bandit_synthesis_claude_sonnet/datasynthesized_programs/

```

```

syn_progs.bandit_self_score.mixed.csv (1766 attacks)
experiment_119_bandit_synthesis_claude_haiku/datasynthesized_programs/
syn_progs.bandit_self_score.mixed.csv (1920 attacks)
experiment_120_bandit_synthesis_gpt3.5/data/synthesized_programs/
syn_progs.bandit_self_score.mixed.csv (1713 attacks)
experiment_121_bandit_synthesis_llama3-8b/data/synthesized_programs/
syn_progs.bandit_self_score.mixed.csv (1725 attacks)
experiment_122_bandit_synthesis_llama3-70b/data/synthesized_programs/
syn_progs.bandit_self_score.mixed.csv (1397 attacks)

```

Detailed logs for each experiment are available under the logs subfolder. These include:

- Program synthesizer logs, including few-shot examples and example pool at the start and end of each iteration.
- HTTP logs from API calls

Program synthesizer logs: from the program synthesizer including few shot examples selected for each iteration and the history of the few-shot example pool ar

4 BENCHMARKING (EXP. 130)

Selected Synthesized Attacks For our benchmarking experiment, we selected the top 10 synthesized attacks for each target model. The selected attacks can be found at the following paths:

```

ROOT/experiments/experiment_130_benchmark/data/synthesized_programs_top_k/
Meta-Llama-3-70B-Instruct.syn_progs.bandit_self_score.mixed.csv
gpt-3.5-turbo.syn_progs.bandit_self_score.mixed.csv
gpt-4o-2024-05-13.syn_progs.bandit_self_score.mixed.csv
Meta-Llama-3-8B-Instruct.syn_progs.bandit_self_score.mixed.csv
claude-3-sonnet-20240229.syn_progs.bandit_self_score.mixed.csv
claude-3-haiku-20240307.syn_progs.bandit_self_score.mixed.csv

# the following attacks from experiment#117
# were also included in the final benchmark
# but not reported in the main results
gpt-4o-2024-05-13.syn_progs.bandit_random.mixed.csv
gpt-4o-2024-05-13.syn_progs.bandit_self_score.lle.csv
gpt-4o-2024-05-13.syn_progs.bandit_offspring_score.mixed.csv

```

Reference SOTA attacks The 23 reference SOTA attacks available at the following path were also included in the benchmark

```

ROOT/experiments/experiment_130_benchmark/
config/sota_programs.csv (23 attacks)

```

Final 113 attacks used for benchmarking The final set of attacks used to benchmark the 6 target models is available here

```

ROOT/experiments/experiment_130_benchmark/data/benchmark/
h4rm3l_benchmark_20240604.csv (113 attacks)

```

AdvBench prompt Samples used for benchmarking The 50 AdvBench that we sampled for benchmarking are available here:

```

ROOT/experiments/experiment_130_benchmark/data/
sampled_harmful_prompts/benchmark-advbench-50.csv (50 prompts)

```

Decorated Prompts The 113 selected attacks were used to decorate each of the 50 prompts, for 5650 decorated prompts available here:

```
ROOT/experiments/experiment_130_benchmark/data/
  decorated_prompts/benchmark-advbench-50.decorated.csv (5650 decorated prompts)
```

Model Responses & Evaluation Each target model was prompted with each of the 5650 decorated prompts. The resulting model responses are available here at the below paths. The CSV files contain the *eval_harmful* column, which contains the output of our harm classifier.

```
ROOT/experiments/experiment_130_benchmark/results/benchmark-advbench-50.decorated
  .evaluated_claude-3-haiku-20240307.csv (5650 responses)
  .evaluated_claude-3-sonnet-20240229.csv (5650 responses)
  .evaluated_gpt-3.5-turbo.csv (5650 responses)
  .evaluated_gpt-4o-2024-05-13.csv (5650 responses)
  .evaluated_Meta-Llama-3-70B-Instruct.csv (5650 responses)
  .evaluated_Meta-Llama-3-8B-Instruct.csv (5650 responses)
```

Qualitative Analysis We clustered synthesized programs, and plotted the average attack success rate achieved as a function of the number of primitives used by each attack results/prog-synthesis-clusters

```
ROOT/experiments/experiment_130_benchmark/results/

# Frequency of each primitive in synthesized attacks > 80% ASR
prog-synthesis-clusters.txt

# One folder per target model
# (only claude-3-haiku-20240307 shown here)
experiments/experiment_130_benchmark/results/prog-synthesis-clusters/
  # Plot of ASR vs number of primitives used by attacks
  claude-3-haiku-20240307.syn_progs.bandit_self_score.mixed/asr_vs_length.png
  # t-sne plot of CodeBert features of attacks >80% ASR
  claude-3-haiku-20240307.syn_progs.bandit_self_score.mixed/cluster_0.8.png
  # t-sne plot of CodeBert features of all attacks
  claude-3-haiku-20240307.syn_progs.bandit_self_score.mixed/cluster.png
```

5 HARMFUL CONTENT CLASSIFIER

Our `is_harmful()` function, used to determine whether a model response constitutes harm is defined in this file.

```
ROOT/
  h4rm3l/src/h4rm3l/harm_evaluation.py
```