USENIX Security ’25 Artifact Appendix: Unsafe LLM-Based Search: Quantitative Analysis and Mitigation of Safety Risks in AI Web Search

Zeren Luo[1*] Zifan Peng[1*] Yule Liu[1] Zhen Sun[1] Mingchen Li[1,2] Jingyi Zheng[1] Xinlei He[1†]

[1]*The Hong Kong University of Science and Technology (Guangzhou)* [2]*University of North Texas*

A Artifact Appendix

A.1 Abstract

This artifact encompasses the full implementation of our risk-mitigation agent as detailed in “Unsafe LLM-Based Search: Quantitative Analysis and Mitigation of Safety Risks in AI Web Search”. It includes a prompt templates for content refinement and malicious URL detection, the Python code for constructing and operating the agent with options for an XGBoost-Detector, PhishLLM-Detector, and our HtmlLLM-Detector. Evaluators can utilize this artifact to refine AIPSEs’ output content using our designed chain-of-thought prompt and classify URLs as benign or malicious using the above three detectors. All code is provided, and executing the basic tests confirms the proper functionality of each component.

A.2 Description & Requirements

This section lists all the information necessary to recreate the experimental setup used to run our artifact. It covers minimal hardware and software requirements, as well as access to our artifact.

A.2.1 Security, Privacy, and Ethical Concerns

The artifact does not perform any destructive operations or disable any security mechanisms on the evaluator’s machine. It does make outbound HTTP(s) requests in two ways:

- **LLM Calls to OpenAI.** The `is_malicious` function and the content-refinement tool both send content or HTML snippets to OpenAI’s API. Evaluators must supply a valid `OPENAI_API_KEY`, and they should be aware that all content sent to the LLM may be logged by OpenAI. No personal or sensitive data beyond URLs should be included.

- **URL Fetches.** When using the HtmlLLM-Detector variant (URL Detector 4), our code fetches remote pages either via `requests` or via Selenium. Evaluators should be mindful that fetching arbitrary URLs can expose them to potentially malicious or untrusted web pages. It is recommended to run URL-fetching in an isolated environment (e.g., a VM or container) if there is any concern about downloading unknown content.

No private or human-identifiable information is requested at any point. The code does not store or transmit any personal user data beyond what is strictly necessary for detecting phishing, malware, or scam content (i.e., the URLs and HTML code themselves). There are no ethical issues with running the provided code, provided the evaluator does not deliberately experiment on known malicious sites in an uncontrolled environment without caution.

A.2.2 How to Access

We share our artifact via a DOI-archived link in Zenodo:

DOI: `10.5281/zenodo.15607878`

URL: `https://doi.org/10.5281/zenodo.15607878`

The artifact includes all the defense code for our paper and a `readme.md` file that details the structure and usage of our defense method. We highly recommend verifying our code through the `readme.md` file.

A.2.3 Hardware Dependencies

The artifact can be evaluated on any standard x86_64 (or ARM64) machine running a recent Linux, macOS, or Windows OS. No GPU is required unless the evaluator wishes to accelerate the XGBoost model training for further experimentation (but all pre-trained weights are provided). Selenium-based crawling requires that a compatible Chrome/Chromium version and the corresponding ChromeDriver are installed. In summary:

- CPU: 64-bit processor (Intel/AMD/Apple Silicon)

- RAM: $\geq$ 4GB

- Disk: $\geq$ 5GB free for code and dependencies

[*] Contributed equally and listed alphabetically.

[†] Corresponding author (xinleihe@hkust-gz.edu.cn).

A.2.4 Software Dependencies

Operating System:
Linux (Ubuntu 20.04+), macOS (10.15+), or Windows 10/11.
Language and Packages:

- `Python 3.10+`
- `pip` (Python Package Installer)
- Required Python Packages:
  - `openai`
  - `langchain`
  - `langchain-openai`
  - `bs4`
  - `python-whois`
  - `pandas`
  - `selenium`
  - `requests`
  - `xgboost`

A.2.5 Benchmarks

None, we are sharing only the defense code, as we mentioned in the open science section.

A.3 Set-up

A.3.1 Installation

1. **Create a Python Virtual Environment (Recommended):**

```
conda create -n testenv python=3.10
conda activate testenv
```

2. **Install All Python Dependencies:**

```
pip install --upgrade pip
pip install -r requirements.txt
```

3. **(Optional) Install ChromeDriver for Selenium** Download the ChromeDriver at `https://developer.chrome.com/docs/chromedriver/downloads`

4. **Set Environment Variables:** Please see the `readme.md` file, without a valid `OPENAI_API_KEY`, the LLM-based detectors and the content-refinement tool will exit with an error.

A.3.2 Basic Test

Please see the `readme.md` file to run the basic test, as the content is not displayed clearly in LaTeX due to formatting issues.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): HtmlLLM-Detector achieves the highest F1 score among the three detectors. This is demonstrated by the experiment (E1) described in Section 8.2 of the paper, with results illustrated in Table 2 and Table 3 of the paper.

(C2): Agent-based defense achieves the highest successful rate compared to Prompt-based defense. This is demonstrated by the experiment (E2) described in Section 8.2 of the paper, with results illustrated in Table 2 of the paper.

A.4.2 Experiments

(E1): Detectors comparison (45 human-minutes + 120 compute-hours (depend on network stability) + 335MB disk):
Preparation: To conduct the comparison experiment, the following steps are executed with precision. Initially, retrieve the essential files as a dataset for PhishLLM, as detailed in Section 6 of the paper, encompassing webpage screenshots, URLs, and HTML source code, with each URL mapped to a distinct folder for storage. Next, establish the experimental environment for PhishLLM by following the guidelines provided in the official repository at "https://github.com/code-philia/PhishLLM". Next, get the test data (Main risk-inclusive response) in Section 6 of the paper.
Execution: Subsequently, execute the PhishLLM method on the collected dataset to obtain results, recording them in a CSV file; Concurrently, apply the HtmlLLM-Detector method using the same dataset and recording them in the same CSV file.
Results: Finally, implement the XGBoost-Detector method through `main.py` and `tools.py`, generating output in JSON format, to derive the confusion matrices for all three detectors and compute the corresponding performance metrics.

(E2): Agent-based and Prompt-based comparison (70 human-minutes + 50 compute-hour (depend on network stability) + 5MB disk):
Preparation: To conduct the comparison experiment, the following steps are executed with precision. Get the test data (Main risk-inclusive response) in Section 6 of the paper.

Execution: Implement the Agent-based approach by executing the script `main.py` with the test data to produce two JSON format files, employing HtmlLLM-Detector and PhishLLM-Detector respectively. Furthermore, separately run `prompt_defense.py` to evaluate the test data and generate an additional JSON format file.

Results: Conduct manual verification by comparing the two JSON format files generated from the Agent-based method with the JSON format files produced by the XGBboost-Detector in experiment E1, alongside the JSON format file generated from independently testing data with `prompt_defense.py`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2025/`.