Artifact for LIMA: Defining, Benchmarking and Detecting Cross-Layer Vulnerabilities in LLM Inference Frameworks

Sanjib Kumar Sen
ssen@islander.tamucc.edu
Texas A&M University-Corpus Christi

Hannah Longoria
hlongoria1@islander.tamucc.edu
Texas A&M University-Corpus Christi

Bozhen Liu
bozhen.liu@tamucc.edu
Texas A&M University-Corpus Christi

1 Abstract

Local Inference Frameworks (LIFs) such as llama.cpp, vLLM, Ollama, and LocalAI enable users to run large language models (LLMs) on local hardware without relying on remote services. However, these frameworks often load community-shared model files that may contain malicious payloads. We define the Local Inference framework–Model Attack surface (LIMA) as the set of vulnerabilities triggered when LIFs process untrusted model artifacts.

To understand the impact of LIMA, we collected 60 publicly disclosed vulnerabilities across four popular open-source LIFs and curated them into a reproducible dataset. Our analysis derives a six-class taxonomy of root causes, showing that LIMA accounts for 42% of studied vulnerabilities and can lead to serious security consequences including heap buffer overflows, path traversal, denial of service, and remote code execution. These findings demonstrate that a single crafted model artifact can compromise an LIF during model load time, before any prompt or inference occurs.

This artifact provides two open-source tools demonstrating the existence and impact of LIMA:

- **LIMABench** – A containerized reproduction framework covering the 60 real-world vulnerability dataset across the four studied LIFs (`llama.cpp`, `vLLM`, `Ollama`, `LocalAI`), including 24 original proof-of-concept (PoC) artifacts (crafted GGUF files, RPC exploit scripts, and API payloads) contributed for advisories that lacked actionable reproduction materials.
- **LIMAScan** – A taxonomy-driven dynamic testing tool that generates payloads from observed vulnerability patterns, tests them against live LIF containers, and detects crashes, panics, information leaks, and successful exploits. LIMAScan detects 13 known CVEs and discovers 7 previously unknown vulnerabilities in the latest stable releases of the four LIFs.

The artifact is publicly available at https://github.com/apace-lab/LIMA and permanently archived on Zenodo at DOI 10.5281/zenodo.20075284.

Pilot Run To quickly validate the artifact, reviewers may run a single PoC and a short LIMAScan test from our GitHub repository:

```
cd vllm/CVE-2024-11041 && ./CVE-2024-11041.sh
cd LIMAScan && ./run_timed_tests.sh
```

2 Overview

This artifact contains the two tools described in the paper. **LIMABench** demonstrates the existence of LIMA by reproducing a curated dataset of vulnerabilities across four LIFs. **LIMAScan** detects vulnerabilities in the latest stable releases by generating taxonomy-driven payloads and testing them against live instances.

The repository is organized by LIF, with one top-level directory per framework. Each directory contains a README and multiple per-vulnerability subdirectories.

- `LIMABench` – Our driver script to reproduce all the 58 real-world vulnerabilities included in LIMABench, which includes the four LIFs from our study, i.e., `LocalAI`, `Ollama`, `vLLM`, and `llama.cpp`.
  - `README`: the detailed instructions (i.e., commands, requirements, expected output) on how to reproduce the 58 vulnerabilities with different configurations (e.g., timeout);
  - `run_all_pocs.sh`: the script to run the 58 vulnerabilities included in LIMABench.
- `LIMAScan` – Our taxonomy-driven dynamic testing tool, which includes:
  - `README`: the detailed instructions on how to reproduce the 13 known CVEs and 7 new vulnerabilities detected by LIMAScan;
  - `generate_all_payloads.py`: generates all test payloads;
  - `payloads/`: crafted GGUF and API payloads;

  - `patterns.md`: extracted patterns from all CVE's across the four LIF repos;
  - `run_full_cross_tests.sh`: full vulnerability testing for LIMAScan;
  - `run_timed_tests.sh`: timed wrapper around `run_full_cross_tests.sh` sections;
  - `run_vllm_tests.sh`: tests patterns from other frameworks against vLLM.
- `llama-cpp`, `LocalAI`, `Ollama`, `vLLM` – each folder contains:
  - `README`: the real-world vulnerabilities included in our study and this artifact are listed here;
  - One subdirectory per vulnerability (typically named by CVE ID), which includes:
    * A `Dockerfile` or `docker-compose.yml`;
    * A script (e.g., `$CVE-ID.sh`) for the corresponding PoC;
    * A README file describing the vulnerability and expected behavior.

3 Requirements and Setup

To successfully reproduce the results from LIMABench and LIMAScan, the following hardware and software requirements are recommended:

- **CPU:** Minimum 4 cores (8+ recommended for parallel container startup). An x86_64 host is required for several `LocalAI` PoCs. Reviewers using Apple Silicon (arm64) should consult the `LocalAI` build documentation[1] for platform-specific caveats.
- **RAM:** Minimum 8 GB, 16 GB recommended. Each LIFs container is configured with a 4 GB memory limit.
- **Disk:** Minimum 15 GB, 25 GB recommended for container images and crafted artifacts.
- **GPU:** Required by the reproduction for `vLLM`. Our evaluation uses an NVIDIA RTX 6000 Ada (48 GB VRAM) on an AMD Ryzen Threadripper PRO 7985WX server. Any CUDA-capable NVIDIA GPU with at least 16 GB VRAM should be sufficient to reproduce the `vLLM` PoCs and the `vLLM` portion of LIMAScan.
- **Software:** Docker $\geq$ 20.10 with Docker Compose, Python $\geq$ 3.9, `curl`, and standard Unix utilities (e.g., `bash`, `grep`, `awk`). The `vLLM` container additionally requires the NVIDIA Container Toolkit[2] for GPU passthrough.
- **Operating System:** Ubuntu 22.04 LTS or other modern Linux distributions are recommended. macOS is partially supported for CPU-only experiments, but GPU-dependent `vLLM` experiments and several LocalAI PoCs may not reproduce correctly.

[1] https://localai.io/basics/build/

[2] https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html

This artifact uses our GitHub repository with the following command:

```
git clone https://github.com/apace-lab/LIMA.git
cd LIMA
```

or the Zenodo repository here.

4 Reproducibility Guide

4.1 LIMABench

LIMABench reproduces 58 vulnerabilities using per-CVE PoC artifacts. Each vulnerability is self-contained within its own directory, including its execution script and environment configuration.

Scope of Reproduction The artifact ships 58 of the 60 vulnerabilities in our dataset as runnable and reproducible PoCs, while the other two vulnerabilities that cannot be shipped as runnable PoCs are documented in the per-product `README.md` files. To be specific, the reproduction of `CVE-2025-46570` requires multi-tenant timing measurements which is infeasible in an isolated single-container environment; and `CVE-2025-1953` targets the `aibrix` [3] plugin with the core vLLM engine, where the plugin's distributed state cannot be run in a self-contained container setup.

4.1.1 Reproduce All Vulnerabilities

To validate the full dataset, the `LIMABench` folder includes a driver script that executes every PoC sequentially and reports aggregate results with the following commands:

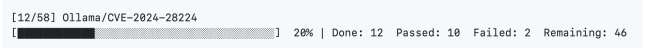
```
cd LIMABench/
./run_all_pocs.sh
```

The default timeout is set to 600 seconds for each PoC, however, reviewers can configure this by the following command:

```
./run_all_pocs.sh --timeout 600
```

which sets the timeout to 10 minutes per PoC.

Expected Output and Execution Time The script displays a live progress bar with the total number of PoCs, the number completed, and running pass/fail counts, as shown below:



Each completed test prints a one-line result as shown below:

```
12/58  Ollama/CVE-2024-28224     CONFIRMED (38s)
```

where the result label is one of following four options:

[3] https://github.com/vllm-project/aibrix

- `CONFIRMED`: Exit 0 and output contains an explicit vulnerability signal (e.g., crash, exploit, info-leak, etc.);
- `SETUP_OK`: Exit 0 but no auto-detectable signal — the environment was set up, however, manual trigger or inspection is needed;
- `FAILED`: Non-zero exit — typically a Docker build/pull error or missing dependency;
- `TIMEOUT`: PoC exceeded the per-test timeout.

A summary, as shown below, will be printed out when the script finishes running all the PoCs, and per-PoC logs are saved at `LIMABench/results/<timestamp>/`:

```
╔═══════════════════════════════════════════════════╗
║              LIMABench — Final Summary            ║
╠═══════════════════════════════════════════════════╣
  Total PoCs run          : 58
  Total elapsed time      : 42m 10s
╠═══════════════════════════════════════════════════╣
  Passed  (exit 0)        : 55   (94%)
    ├─ Confirmed          : 50
    └─ Setup only         :  5
  Failed  (error/timeout) :  3
╚═══════════════════════════════════════════════════╝

Result: 58 issues in LIMABench, 55 successfully reproduced, 3 failed.
```

The full suite takes 1.5–2.5 hours on a cold machine when there is no cached image, and 20–40 minutes on a warm machine when all required images are cached.

4.1.2 Reproduce A Vulnerability

Each PoC can be executed independently. For example, reviewers can run the vulnerability with `CVE-2024-11041` from `vLLM` with the following command:

```
cd vllm/CVE-2024-11041
./CVE-2024-11041.sh
```

Each run script is responsible for:

- Building or launching the required container;
- Supplying the crafted malicious input (e.g., GGUF file or API request);
- Triggering the vulnerability.

Expected Output and Execution Time Each PoC produces output specific to the vulnerability being tested (e.g., crash logs, panic messages, or abnormal responses). The expected behavior and indicators of successful reproduction are documented in the README within each CVE directory.

Reviewers should consult the per-CVE README to interpret the output and map it to the taxonomy categories described in the paper (e.g., crash, panic, information leak, or successful exploit).

Each PoC script is expected to run in approximately 1-4 minutes, depending on whether required dependencies and container images are already available locally.

4.2 LIMAScan

LIMAScan dynamically tests LIF instances using taxonomy-derived payloads through the following steps:

1. Generates payloads based on observed vulnerability patterns
2. Sends payloads to live LIF endpoints (file loading or API interfaces)
3. Monitors responses and logs for abnormal behavior

4.2.1 Running LIMAScan

To reproduce the 13 known CVEs and 7 previously unknown vulnerabilities detected by LIMAScan in the latest stable releases of the four LIFs, reviewers can run the following commands:

```
cd LIMAScan
./run_full_cross_tests.sh
```

which will launch individual LIF containers as needed during the detection.

The following command will reproduce the content in Table 6 from the original paper, including the per-framework execution time and the detected vulnerabilities with details.

```
./run_timed_tests.sh
```

To run LIMAScan against vLLM (GPU required), reviewers can run the following command:

```
cd LIMAScan
./run_vllm_tests.sh
```

Expected Output and Execution Time For each PoC, LIMAScan prints a one-line result indicating the observed behavior (e.g., crash, panic, or normal execution) as shown below:

```
[CRASHED] Heap Overflow/p2a_large_n_kv →
 ↪ LocalAI: OOMKilled=true exit=137 — GGUF
 ↪ triggered fatal OOM
[PANIC_RECOVERED] Integer
 ↪ Overflow/p1a_negative_key_length →
 ↪ LocalAI: Go panic caught by recover()
[NOT_VULNERABLE] Resource
 ↪ Exhaustion/p11b_huge_dims → LocalAI:
 ↪ Server survived
```

where the result label is one of following four options:

- `CRASHED`: Server died (e.g., OOMKilled, non-zero exit, or health check failed);
- `PANIC_RECOVERED`: Go runtime panic caught by `recover()` means server survives but parser is unsafe;
- `VULNERABLE`: Explicit vulnerability signal confirmed (e.g., info leak, auth bypass, etc.);

- `ERROR_RETURNED`: Server returned an HTTP error (e.g., `5xx`) on the crafted input;
- `NOT_VULNERABLE`: Server handled the payload without abnormal behavior;
- `SKIPPED`: Container failed to start or prerequisite environment was missing.

At the end of execution, LIMAScan outputs a reference to a result file containing detailed logs, including per-framework execution time and observed outcomes. A summary across all tested frameworks is also displayed, including the total numbers of tests and counts of each outcome category as shown below:

```
Total tests: 23
  CRASHED: 2
  PANIC_RECOVERED: 3
  VULNERABLE: 1
  NOT_VULNERABLE: 17
  SKIPPED: 0
```

Reviewers should observe abnormal outcomes (e.g., crashes or panics) indicating the detected vulnerabilities. In total, LIMAScan should detect 13 known vulnerabilities and identify 7 previously unknown vulnerabilities, consistent with our paper.

Running both the full scan and timed scan on the four LIFs costs approximately 4 minutes. The execution time of running LIMAScan against vLLM requires approximately 3-5 minutes depending on GPU availability and model load time.

5 Evaluation Criteria and Expected Results

Artifact reviewers should evaluate the artifact based on the following criteria:

1. Successful Execution

- Docker containers build and start correctly
- LIMABench and LIMAScan execute without runtime errors

2. Reproducibility of Vulnerabilities

- Approximately 58 vulnerabilities should reproduce successfully
- Observed outputs match expected behaviors (e.g., crashes, panics)

3. Detection Capability

- LIMAScan re-discovers 13 known vulnerabilities
- LIMAScan detects 7 previously unknown vulnerabilities

4. Output Interpretation LIMABench and LIMAScan do not aggregate results into a single counter by default. Instead, each test prints a standardized result line indicating its outcome.

For LIMABench, each PoC produces one of the following outputs:

- `CONFIRMED`
- `SETUP_OK`
- `FAILED`
- `TIMEOUT`

Reviewers should interpret each occurrence of the above outputs as a successful reproduction of a vulnerability. Failed reproductions correspond to cases where no such signal is observed.

For LIMAScan, each payload execution prints:

```
[RESULT] Pattern/test → Framework: detail
```

where `STATUS` is one of: `CRASHED`, `PANIC_RECOVERED`, `VULNERABLE`, `ERROR_RETURNED`, `NOT_VULNERABLE` or `SKIPPED`.

A successful detection is indicated by any status other than `NOT_VULNERABLE`.

Expected Counts Based on our evaluation in the paper:

- LIMABench should reproduce approximately 58 out of 60 vulnerabilities
- LIMAScan should detect:
  - 13 known vulnerabilities
  - 7 previously unknown vulnerabilities

Reviewers can verify these numbers by counting the occurrences of the corresponding output labels in the console logs.

5. Coverage

- All four LIFs are exercised: `llama.cpp`, `vLLM`, `Ollama`, `LocalAI`
- Both reproduction (LIMABench) and detection (LIMAScan) workflows are validated

Limitations

- vLLM experiments require GPU support and may not run on CPU-only systems
- Some PoCs require x86_64 architecture
- Two vulnerabilities cannot be reproduced due to environmental constraints (as described in the paper)