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

Abstract

Local Inference Frameworks (LIFs) such as llama.cpp, vLLM, Ollama, and LocalAI enable users to run large language models on their own hardware, avoiding data exposure to remote services. However, these frameworks often load community-shared model files that can carry malicious payloads. We define the LIF-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 the above four popular open-source LIFs, resulting in a curated dataset for our study. We analyzed their root causes and derived a six-class taxonomy, where LIMA accounts for 42% and leads to serious attacks including heap buffer overflows, path traversal, and even remote code execution. This demonstrates that a single crafted model file can attack during model load time, before any user prompt or inference takes place. We further build LIMABench, an automated reproduction framework for our collected dataset for consistent verification. To systematically uncover LIMA vulnerabilities, we develop LIMAScan, a taxonomy-driven dynamic testing tool that generates LIMA-pattern-derived payloads and tests them against live LIF instances to detect unsafe GGUF metadata handling patterns, through which we discover and successfully exploit 7 previously unknown vulnerabilities in the latest stable releases of the four LIFs. We release LIMABench and LIMAScan as an open-source artifact.

1 Introduction

Large language models (LLMs) have been extensively integrated into open-source software [15,26], where LLMs are the core component that enables the primary functionality instead of acting as an auxiliary component, named LLM-powered applications (LPAs). According to a recent study [48], around 61% popular open-source LPAs support LLM inference on local machines and private infrastructure through open-source platform or libraries, such as `llama.cpp` [4] and `Ollama` [22].

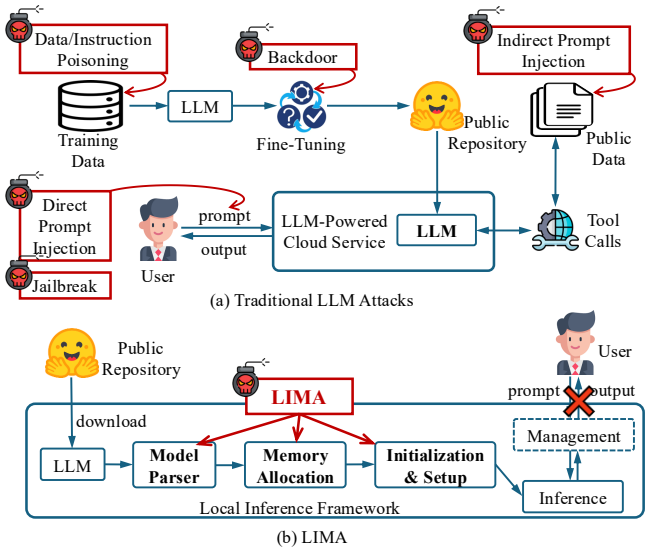

Figure 1: LIMA vs. Traditional Malicious LLM Attacks.

We use the term **L**ocal **I**nference **F**rameworks [47] (**LIF**s) to denote such a growing class of open-source systems designed to run LLMs directly on end-user or developer hardware, which emphasize efficient loading, quantization, and inference on local devices.

The primary design goal of using LIFs in LPAs is to preserve end-user data safety and privacy by performing inference locally, rather than transmitting data to remote LLM services through APIs. In addition, LIFs offer users the flexibility to select from a wide range of open-source LLMs tailored to different tasks and purposes, where users typically download models from public LLM ecosystems such as Hugging Face [5]. However, this flexibility and the perceived "data protection by locality" expose a new attack surface, where malicious or tampered model files obtained from open-source LLM ecosystems (*e.g.*, Hugging Face [5]) can compromise the systems intended to safeguard user data.

In this paper, we define the **L**ocal **I**nference framework-

Model **A**ttack surface (**LIMA**) as the set of vulnerabilities arising from the interaction between LIFs and the model artifacts they load. Unlike prior malicious model attacks that operate at inference time, such as prompt-based manipulation (*e.g.*, jailbreak [41] and prompt injection [17]) or behavioral tampering (*e.g.*, backdoors [69] and poisoning [36,40]), LIMA targets vulnerabilities in the model loading and initialization pipeline (including parsing, memory allocation and handling) as illustrated in Figure 1, enabling exploitation before any inference occurs and independent of model semantics. This attack surface emerges from the unique design of LIFs, which prioritize performance and compatibility over strict input validation, and routinely ingest community-shared models from open-source ecosystems.

To better understand LIMA, we identified four widely used LIFs (including `llama.cpp`, `vLLM`, `Ollama` and `LocalAI`) by surveying popular open-source LPAs [48] and analyzing their runtime dependencies. For these LIFs, we collected their publicly disclosed security vulnerabilities from the National Vulnerability Database (NVD), resulting in a curate dataset of 60 LIF vulnerabilities that form the foundation of our study. We then identify their root causes, vulnerability patterns and security impact by systematically analyzing their vulnerable code snippets, patches and documentations against the exploitation. Based on these findings, we derive a six-class taxonomy grounded in the observed root causes and exploitation patterns, where LIMA attacks accounts for 42% and leads to heap buffer overflows, denial of service, and even remote code execution (RCE). Unsafe deserialization and LIMA together account for 53%, both of which are attributed to unconditional trust in model-controlled values without validation. Notably, the same GGUF header fields that cause heap overflows in C/C++ (*e.g.*, `llama.cpp`) also lead to runtime panics in Go (*e.g.*, `Ollama`). This demonstrates that even though the programming languages may affect the crash manifests, they do not eliminate the underlying security flaw.

To enable systematic validation and reproduction, we build **LIMABench**, an automated reproduction framework built on our LIF vulnerability dataset. LIMABench is empowered with our reproduction workflows, enabling consistent benchmarking and regression-testing of LIMA vulnerabilities. We contribute 24 original proof-of-concept (PoC) artifacts including crafted GGUF files, RPC exploit scripts and API payloads. Although LIMABench includes both LIMA and other vulnerabilities collected from LIFs, its primary goal is to benchmark and reproduce LIMA vulnerabilities.

To effectively identify LIMA, we design and implement **LIMAScan**, an automated taxonomy-guided dynamic testing tool that generates crafted payloads from our observed vulnerability patterns and tests them against live LIF instances to detect exploitable flaws observed from our study. LIMAScan targets at the semantic interaction between model metadata and its handling logic in LIFs. By analyzing metadata-flow paths and configuration-driven behaviors, LIMAScan uncovers recurring design and implementation flaws that can be exploited to trigger LIMA. We applied LIMAScan to the most recent stable releases of the four popular LIFs, revealing 13 previously disclosed CVEs that remain unpatched. Interestingly, LIMAScan also identified and successfully exploited 7 previously unknown vulnerabilities, demonstrating the potential of LIMAScan and real-world impact of LIF vulnerabilities. We have reported these new vulnerabilities to the corresponding developers through GitHub security advisories.

In summary, this paper has the following contributions:

- We define LIMA, the Local Inference framework-Model Attack surface, a new load time attack class arising from maliciously crafted model artifacts in local LLM runtimes.
- We construct a dataset of 60 publicly disclosed LIF vulnerabilities and perform a systematic analysis, deriving a taxonomy of vulnerability patterns with their root causes.
- We build LIMABench powered by an automated reproduction framework for our dataset to enable consistent verification and inspire future defensive approaches.
- We develop LIMAScan, an automated taxonomy-guided dynamic testing tool which detects 7 previously unknown vulnerabilities from the four popular LIFs.
- We release LIMABench and LIMAScan publicly [1] to support reproducibility and future research.

2 Background and Threat Model

2.1 Traditional Model Threats

Malicious LLMs are models intentionally developed or modified to bypass safeguards and produce harmful or illegal outputs, which are either deliberately configured without restrictions or optimized to assist activities such as phishing and exploit development [10]. Figure 1(a) highlights existing malicious LLM threats, which can be grouped into four levels.

At the prompt level, open-released or weakly aligned models can be easily reconfigured for harmful use through jailbreaking [41, 67] or direct prompt injection [17, 45]. Existing work has shown that aligned models remain vulnerable to adversarial jailbreak attacks that bypass safety controls [64, 74]. In this case, the model itself is not only misused but also deployed without safeguards.

At the context level, the model's execution context can be manipulated through indirect prompt injection [50, 72] and retrieval data poisoning [40, 75]. These attacks exploit auxiliary components such as retrieval-augmented systems, external knowledge bases or tool integrations, which introduce attacker-controlled content into the model's input context at inference time. In such cases, the model itself is safe, and the vulnerability arises from the surrounding system components that mediate its inputs.

[1] https://github.com/apace-lab/LIMA

At the model level, attackers can intentionally implant hidden malicious behaviors through training data poisoning [36,51], weight poisoning [55,57] or backdooring [69–71]. Prior work has demonstrated that LLMs can be trained to behave safely under normal evaluation while activating harmful behavior under specific triggers [52]. Other work also shows that malicious behaviors can be inserted efficiently through parameter editing, without requiring large-scale retraining or obvious data poisoning [58]. In such scenarios, the LLM itself becomes a compromised artifact that appears benign but contains intentionally embedded attacks.

At the system level, malicious code or harmful behavior can also be embedded within components of the LLM pipeline, where the LLM-integrated system enables a broader attack workflow [61]. For example, retrieval components can be manipulated to control downstream outputs [39]. When combined with automation tools, APIs, or malware toolkits, such malicious or compromised LLM pipelines can scale phishing campaigns or assist in vulnerability discovery.

Overall, the above malicious LLMs primarily enable harmful behavior through deliberate prompt design, data tampering, weight modification or by intentionally disabling safeguard mechanisms to allow unrestricted interaction during user chat. These models are intentionally configured to produce harmful outputs as a core function, which are fundamentally and structurally different from the malicious models we observed from security vulnerabilities in LIFs.

2.2 Local Inference Frameworks

We define **L**ocal **I**nference **F**rameworks (**LIFs**) as a class of open-source systems designed to enable execution of LLMs on local or self-hosted environments without reliance on remote inference services. As shown in Figure 2 (b), LIFs typically provide one of the following functions: (1) low-level inference engines, such as `llama.cpp` and `vLLM`, which implement model loading, tensor parsing, memory management and token generation; and (2) management layer, such as `Ollama` and `LocalAI`, which manage model distribution, configuration, authentication, APIs and proxy to underlying inference engines.

Unlike large-scale serving infrastructures such as TensorFlow Serving [14] or NVIDIA Triton [9], which target distributed deployments and cloud datacenters, LIFs are lightweight, flexible, and often adopted in open-source end-user applications [47, 48]. Although these frameworks differ in all dimensions (*e.g.*, architecture, purposes and implementation), they share the ability to ingest and initialize arbitrary community-shared models, combined with performance-driven parsing of model formats, makes them both popular and uniquely exposed to security risks. This shared model-loading path forms the foundation of LIMA.

2.3 New Threat Model in LIFs

Following the red edges in Figure 2, we consider an adversary who distributes a malicious or tampered model file intended to be loaded by an LIF. The attacker may upload this model to a public repository (*e.g.*, Hugging Face) or share it through community channels where users commonly obtain open-source LLMs. The victim is an end user or developer who downloads and runs the model locally, believing it to be safe. Once the model is parsed or initialized, the malicious payload can exploit vulnerabilities in the LIF's model loading pipeline, such as unchecked metadata, unsafe memory allocations, or integer overflows, to trigger crashes, corrupt memory, or achieve arbitrary code execution. The attacker's goal is to compromise the confidentiality or integrity of the user's local environment, potentially exfiltrating data or establishing persistent control.

This paper focuses exclusively on vulnerabilities arising from the interaction between LIFs and the loaded model artifacts, which is referred to as LIMA in this paper. We analyze such attacks that occur during load time preparation, before any user prompts or inference (during runtime) take place. Other attack vectors in the LLM and machine learning ecosystem such as the scenarios we discussed in Section 2.1 or remote inference service exploits, are outside the scope of this study. Our analysis is limited to open-source frameworks and model formats commonly used for local LLM inference (*i.e.*, GGUF). We also assume that the attacker has not already gained control over the victim's machine or underlying hardware and that sandboxing, if present, provides standard process-level isolation.

3 The LIF–Model Attack Surface (LIMA)

In this section, we officially define the new attack surface, illustrate the attack pipeline with the parsed models behave as untrusted structured inputs.

3.1 Definition of LIMA

LIMA includes vulnerabilities that arise when an LIF ingests and initializes externally supplied model artifacts, where attacker-controlled model metadata and tensor parameters affect parsing, allocation and/or initialization logic prior to the processing of prompts and execution of inference.

3.2 Models as Untrusted Structured Inputs

LIFs often run models downloaded from public ecosystems such as Hugging Face, where users assume model artifacts are safe. Existing research has shown that publicly distributed models can embed backdoors and poisoned weights, highlighting the risks associated with using those models [6, 33].

To detect malicious models before publication, major public model distribution platforms have introduced safety mechanisms such as automated scanning [7, 11, 44] and restricted

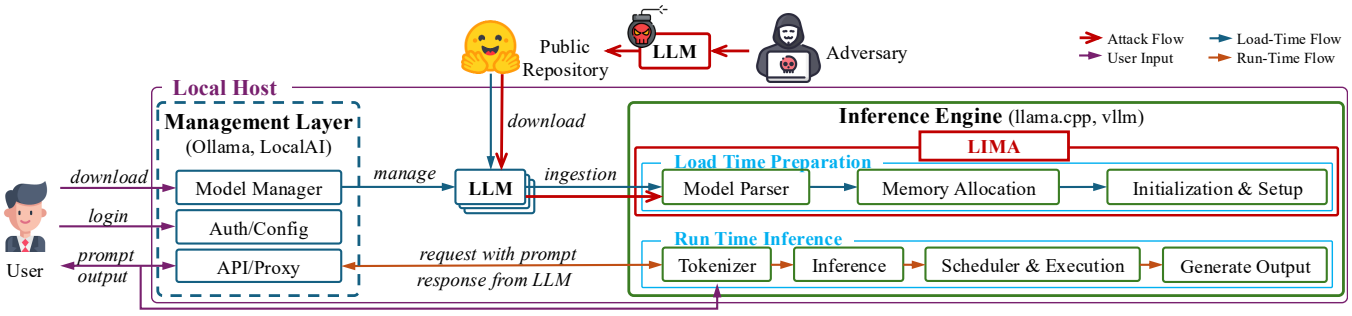

Figure 2: Architecture and Threat Model of LIFs. The LIF deployments may include an optional management layer that exposes APIs and orchestrates model loading, or directly invoke an inference engine. User prompts are processed at runtime, whereas model artifacts are ingested at load time. The LIMA resides at this model ingestion boundary.

Table 1: Mapping between GGUF ingestion pipeline, stages, attacker-controlled fields, and common bug classes in LIMA.

Pipeline	Stage	Attacker-Controlled Fields	Common Errors
Model Parser	Parse header	`n_kv`, `n_tensors`	Integer overflow, Heap corruption, DoS
	Parse metadata	`key_len`, `str_len`, `value_type`, `n_elems`	OOB read/write, Overflow, DoS
	Parse tensor directory	`name_len`, `n_dims`, `dims[]`, `dtype`, `offset`	OOB read/write, Overflow, NULL deref, DoS
Memory Allocation	Compute allocation size	`n_kv`, `n_tensors`, `n_dims`, `dims[]`, `dtype`	Integer overflow, Heap corruption
	Allocate memory	`n_dims`, `dims[]`, `dtype`	Heap overflow, DoS
Initialization	Initialize runtime structures	Metadata keys/values, `n_tensors`, `name_bytes`, `dtype`	Logic errors, OOB access
	Copy / map tensor data	`offset`, tensor size (from `dtype` + `dims[]`)	OOB read/write, Heap corruption, Divide-by-zero, RCE

serialization formats [54]. However, these protections primarily focus on malicious model behavior or unsafe serialization patterns, not vulnerabilities in downstream LIFs. Even though a model is semantically benign and passes safety checks, its metadata and structural fields can remain attacker-controlled and crash LIF's parsing and initialization logic. Hence, safety checks from the distribution platforms cannot eliminate vulnerabilities arising from load time handling in LIFs, leaving the model ingestion pipeline exposed.

3.3 Model Ingestion Pipeline

Model ingestion in LIFs often follows a structured pipeline with several stages as shown in columns 1 and 2 of Table 1: the framework imports the model artifact (in GGUF format), parses its header and metadata fields, reads tensor dictionary, computes allocation size for later memory allocation, and initializes runtime data structures to get ready for running inference requests. These steps include intensive arithmetic operations fully relying on model-controlled fields (*e.g.*, tensor dimensions, counts, offsets), which determines the dynamic heap allocation size on memory.

Note that the entire pipeline ingests model artifact before any user prompt is processed. Hence, vulnerabilities in the pipeline can be triggered only by loading a maliciously crafted model file. Based on our study, GGUF and Pytorch are supported by the four LIFs. vLLM uses Pytorch and safetensor [13], while the other three LIFs have direct support for GGUF.

3.4 Attacker-Controlled Fields

A GGUF file contains numeric fields that the LIF parser uses directly in array indexing operations and function calls such as `malloc` and `memcpy`. Since there is no bounds checking, an attacker can control these fields and trigger LIMA attacks. During the ingestion pipeline, attackers control multiple structurally significant fields in the model artifacts, such as tensor counts `n_tensors` and dimension sizes `n_dims` as shown in column 3 of Table 1. Moreover, some frameworks also expose file paths and configuration fields during the import process. These values are directly consumed by parsing and memory allocation logic to determine critical steps for initialization and runtime behavior, such as memory layout and allocation size. Since allocation sizes and buffer boundaries are frequently derived from these fields, carefully crafted values can induce integer overflows, out-of-bounds (OOB) accesses, or inconsistent state transitions during initialization, indicated by column 4 of Table 1. Moreover, model registries add more attack vectors at the distribution layer when digest strings without validation are used in file path construction or when authentication challenges redirect credentials to attacker-controlled servers. We observed the above representative patterns across all four LIFs we studied.

In summary, instead of manipulating complex model semantics or inference outputs, attackers can easily perform an

exploitation by injecting structurally inconsistent fields between model-controlled metadata and implementation-level assumptions. This motivates the design of our dynamic testing tool in Section 5.

3.5 Compare LIMA with Other Model Attacks

Table 2 compares representative LLM attacks across modified content, attack phase, target layer, required adversarial control and persistence. As we discussed in Section 2, traditional attacks primarily aim at three targets: (1) input-level attacks manipulate inference time prompts without modifying model parameters, such as jailbreaking and direct prompt injection; (2) context-level attacks affect model behavior by injecting adversarial content or behavior through auxiliary components (*e.g.*, retrieval database or external tools), such as indirect prompt injection and retrieval poisoning [39, 75] (*i.e.*, one type of data poisoning); (3) model-level attacks modifies training data or parameters to induce persistent behavioral changes, such as backdooring, data and weight poisoning. Differently, LIMA operates at the LIF layer, which does not rely on malicious prompts or complex manipulation of model weights. Instead, LIMA exploits unsafe model ingestion and parsing logic during load time through crafted model artifacts. These emphasize LIMA as a new attack surface and patterns which targets runtime integrity of LIFs instead of model behavior or context inputs.

4 Empirical Study of LIF Vulnerabilities

In this section, we introduce our study on publicly released LIF vulnerabilities, discuss the vulnerability patterns of LIMA and other attacks, and conclude a taxonomy based on their root causes.

Data Collection We first identified widely adopted LIFs by surveying open-source LPAs [48] and examining their documented runtime dependencies and deployment instructions. This process yielded a set of representative frameworks as shown in column 1 of Table 3, including inference engines (*e.g.*, `llama.cpp`, `vLLM`) and management layers (*e.g.*, `Ollama`, `LocalAI`). Column 2 lists their numbers of stars and forks, indicating their popularity and wide adoption. For each selected framework, we systematically queried NVD using the framework name as keyword. We then manually inspected each retrieved CVE or GHSA entry to confirm relevance to LIF operation, examining linked advisories, patch commits and issue trackers.

Automated Reproduction Out of 60 collected vulnerabilities, 36 were accompanied by publicly disclosed PoC artifacts such as crafted model file, exploit scripts, or detailed reproduction commands provided by the original reporters. The remaining 24 advisories provided only conceptual vulnerability descriptions without actionable reproduction materials; for these, we developed original PoC artifacts including crafted GGUF files, RPC exploit scripts and API payloads. We then construct an automated reproduction framework based on these PoCs that standardizes the procedures triggering vulnerabilities across affected LIF versions. Where necessary, we adapt or minimize existing PoCs to ensure reproducibility and environmental consistency.

We successfully containerize 58 reproduction environments and release our automation framework with our curated dataset (including 60 LIF vulnerabilities) as an open-source artifact to support independent verification and future research. Two vulnerabilities from `vLLM` could not be reproduced in isolated container environments cannot be explored, because: (1) CVE-2025-46570 [30] requires multi-tenant timing measurements infeasible in isolated containers, and (2) CVE-2025-1953 [27] targets the `aibrix` plugin together with the core vLLM.

Data Analysis For each vulnerability in our dataset, we examined the associated CVE and/or GHSA entry, advisories, patch commits, and reproduction artifacts to determine its root cause, trigger conditions, and affected components within the corresponding LIF. We analyzed how model-controlled inputs, such as tensor dimensions, metadata fields, file paths, or configuration parameters, interact with parsing, allocation and initialization logic to induce failure. We further mapped each vulnerability to the specific source modules involved, enabling identification of root causes shared components across frameworks. Based on the above analysis, we derive a taxonomy of recurring vulnerability patterns within LIF model ingestion pipeline.

To ensure the rigor and reliability of our study, we adopted a structured classification process [42] when deriving the taxonomy. All authors independently reviewed and labeled each vulnerability according to its primary root cause, triggering condition and vulnerable code snippet. We compared the independent labels and resolved disagreements through discussion until consensus was reached. Instead of imposing predefined categories, we derived taxonomy classes inductively based on recurring patterns observed across LIFs, refining definitions iteratively as new patterns emerged. This process helps minimize subjective bias and improve consistency in our study.

Next, we will report our analysis result, including the observed recurring vulnerability patterns, the root cause analysis and our derived taxonomy.

4.1 Landscape of LIF Vulnerabilities

Our analysis covers 60 vulnerabilities across four LIFs as illustrated in Table 3: `llama.cpp` (15), `vLLM` (25), `Ollama` (15), and `LocalAI` (5). Among these, 16 are rated critical severity, 25 high, 15 moderate, and 4 low, with the critical and

Table 2: Comparison of LLM attacks across different dimensions*.

Attack	Modified Content (Modify Weights)	Attack Phase	Primary Target Layer	Require Malicious Model	Require Malicious Input	Persistence
Jailbreaking	User Prompt (×)	Inference Time	Input Layer	×	✓	No
Direct Prompt Injection	User Prompt (×)	Inference Time	Input Layer	×	✓	No
Indirect Prompt Injection	Retrieved/Tool-Generated Content (×)	Inference Time	Context Layer	×	✓	No
Data Poisoning	Training Dataset (✓)	Training Time	Context/Model Layer	✓	×	Yes
Backdooring	Training Dataset (✓)	Training Time	Model Layer	✓	×	Yes
Weight Poisoning	Model Parameters (✓)	Supply Chain	Model Layer	✓	×	Yes
LIMA	***Model File Structure***, ***Metadata*** (×)	***Load Time*** (Parse, Alloc, Init)	***LIF***	✓ (Crafted Model File)	×	Yes

* The entries in each cell represent major or representative mechanisms of the corresponding attack and are not intended to be exhaustive.

Table 3: Statistics and vulnerability distribution of the popular LIFs in our study.

		#CVE						
LIF	**#Star/#Fork**	**C1***	**C2**	**C3**	**C4**	**C5**	**C6**	**Total**
llama.cpp	96.4k/15.2k	12	0	3	0	0	0	15
vLLM	71.7k/13.9k	2	7	1	11	4	0	25
Ollama	164k/14.7k	11	0	1	3	0	0	15
LocalAI	43.2k/3.6k	0	0	0	3	0	2	5
Total	-	**25**	**7**	**5**	**17**	**4**	**2**	**60**

* C1 aligns with LIMA.

high severity vulnerabilities concentrated in model file parsing and unsafe deserialization. As shown in Table 4, we organize these vulnerabilities into six classes and 21 sub-classes based on their root cause mechanism. The table also reports the number and severity of CVEs in each class, the affected LIFs and the key mechanisms that trigger these vulnerabilities.

In summary, the largest class is malicious model file attacks (C1, 25 CVEs, 42%), followed by API input validation failures (C4, 17 CVEs), unsafe deserialization in distributed communication (C2, 7 CVEs), exposed network services (C3, 5 CVEs), cache poisoning and side-channel leakage (C5, 4 CVEs), and web interface security gaps (C6, 2 CVEs). While classes C2 through C6 involve vulnerability patterns known from traditional software (without LLM integration), they manifest in contexts specific to LLM inference such as grammar compilation engines, multi-tenant prefix caching, and distributed tensor parallelism. This indicates how traditional software weakness can acquire new security implications in modern AI systems, especially in LIF-integrated pipelines.

Table 3 also summarizes the distribution of CVEs categorized as LIMA vulnerabilities across the six taxonomy classes for each LIF. Column 3-8 report the numbers of CVEs in each class with the total number listed in column 9, highlighting how vulnerability patterns and root causes can vary across different designs and implementations.

LIMA Patterns Malicious model file attacks (C1) are the most distinctive class across these LIF vulnerabilities, which align directly with LIMA defined in Section 3. This class contains four sub-classes: memory corruption through GGUF parsing in `llama.cpp` (11 CVEs), denial of service (DoS) through crafted metadata in `Ollama` (7), code execution through embedded model content such as Jinja2 templates and pickle payloads (3), and file system attacks through model registry workflows in `Ollama` (4). The same untrusted GGUF header fields that cause heap buffer overflows in C/C++ also trigger runtime panics in Go, proving that the programming languages affect the vulnerability types but do not eliminate the underlying validation problem.

Other Attack Patterns Unsafe deserialization (C2, 7 CVEs) is concentrated entirely in `vLLM`, where every distributed communication channel uses Python's `pickle` serialization, including ZeroMQ sockets, shared memory ring buffers, and PyTorch's TCPStore. Each instance enables remote code execution, making pickle a systemic architectural vulnerability rather than a collection of isolated bugs. This highlights a broader challenge in open-source ecosystems: even though `pickle`'s official documentation clearly warns that its insecure serialization can execute arbitrary code during deserialization [12], developers continue to use it in unsafe contexts without sufficient validation or safeguards.

API input validation failures (C4, 17 CVEs) cover the widest range of patterns, from structured output engine crashes where invalid JSON schemas or regex patterns crash the grammar compilation backend, to command injection through unsanitized configuration parameters in `LocalAI`. Several of these vulnerabilities were first disclosed in 2025, suggesting that structured output and guided decoding represent an emerging attack surface as LLM inference matures.

Table 4: Taxonomy of 60 LIF vulnerabilities organized by root cause mechanism

Class (#Total)	Sub-Class	#CVEs	Severity	LIFs	Key Mechanism
C1: Malicious Model File Attacks (25) ***(LIMA)***	C1.1: Memory corruption via parsing	11	Mod.–Crit.	llama.cpp	Heap overflow, integer overflow, OOB, null ptr
	C1.2: DoS via crafted metadata	7	High	Ollama	Divided-by-zero, unbounded alloc, truncated GGUF
	C1.3: Code exec via model content	3	High–Crit.	vLLM, llama.cpp	Jinja2 SSTI, `torch.load()`, pickle
	C1.4: File system via registry	4	Med.–Crit.	Ollama	Path traversal, file deletion, token theft
C2: Unsafe Deserialization (7)	C2.1: Direct pickle usage	6	High–Crit.	vLLM	ZMQ, shared memory, TCP, `pickle.loads()`
	C2.2: Framework abstractions	1	Critical	vLLM	PyTorch TCPStore internal pickle
C3: Exposed Network Services (5)	C3.1: Unauthenticated RPC	3	Mod.–Crit.	llama.cpp	Arbitrary memory read/write via raw pointer
	C3.2: All-interface binding	1	High	vLLM	ZMQ XPUB on `tcp://*`
	C3.3: Missing access controls	1	Medium	Ollama	DNS rebinding bypasses localhost API
C4: API Input Validation Failures (17)	C4.1: Structured output crashes	3	Moderate	vLLM	Invalid schema/regex crashes xgrammar
	C4.2: ReDoS	2	Moderate	vLLM	Backtracking regex in tool parsers
	C4.3: Algorithmic complexity	1	Moderate	vLLM	$O(n^2)$ list concat in tokenizer
	C4.4: Resource consumption	4	Med.–High	vLLM, Ollama	`best_of`=500, 500 MB header, `/dev/random`
	C4.5: Unbounded caching	2	Moderate	vLLM	xgrammar RAM + outlines disk, no eviction
	C4.6: Info disclosure	2	High	Ollama	`os.Stat` errors reveal FS state
	C4.7: Command/code injection	2	High–Crit.	LocalAI	Config params being executed by `exec.Command()`
	C4.8: SSRF / LFI	1	Medium	LocalAI	`file://` and `http://` unvalidated
C5: Cache Poisoning / Side-Channel (4)	C5.1: Predictable hash collisions	3	Low–Mod.	vLLM	PIL tobytes, `hash(None)`, fixed seed
	C5.2: Timing side-channel	1	Low	vLLM	Shared prefix cache leaks prompt via TTFT
C6: Web Interface Security Gaps (2)	C6.1: Cross-site scripting	1	Medium	LocalAI	Stored XSS via unsanitized model name
	C6.2: CSRF	1	Medium	LocalAI	No tokens on state-changing endpoints
Total		**60**			

4.2 Root Cause Analysis

The six classes describe different attack vectors, however many of them share common underlying failure patterns. In addition to the attack mechanisms, we examined *why* these vulnerabilities exist by analyzing their root causes and identified four root cause categories that appeared independently across multiple frameworks.

Table 5 illustrates the four root causes account for 56 of 60 vulnerabilities (93%) across the four LIFs in our study. The remaining 4 vulnerabilities belong to cache poisoning and side-channel leakage (C5), which appear only in `vLLM` and do not generalize across frameworks as they require multi-tenant serving with shared caches, a feature that the other three frameworks currently lacks.

Unsafe Model File Parsing (18 CVEs, 30%) One of the common pattern we have found is that GGUF model file parsers read values from file headers directly (*e.g.*, string lengths, key-value (KV) pair counts, tensor dimensions, alignment values) and use them in memory operations without validation, leaving a serious attack surface in model loading. In `llama.cpp`, which is written in C/C++, the unvalidated header fields lead to heap buffer overflows (5 CVEs), integer overflow in size calculations (3 CVEs), null pointer dereferences (1 CVE), OOB array access (1 CVE), and use of uninitialized memory (1 CVE). In `Ollama`, which is written in Go, the same type of unvalidated fields trigger division-by-zero panics (2 CVEs), unbounded memory allocation (2 CVEs), and crashes from truncated or malformed model data (3 CVEs).

Go's memory safety prevents buffer overflows that occur in C/C++, but does not prevent the runtime panics that crash the server. Our dynamic testing tool (LIMAScan) confirmed this pattern also affects `LocalAI`: the same crafted GGUF file with 4 billion KV pairs directs the `gpustack/gguf-parser-go` [18] library to attempt an unbounded allocation, which triggers an OS-level out-of-memory kill. We identify this crash as a previously undiscovered issue, which we denote as LIMA-NEW-007 and discuss in detail in Section 7.

Untrusted Code Execution (12 CVEs, 20%) Another common root cause is that user-controlled input (*e.g.*, model files, network data, configuration parameters) is passed into code execution paths without sanitization or sandboxing. This root cause manifests through four different mechanisms depending on the framework and its programming language.

In `vLLM` (written in Python), the major cause is the use of unsafe pickle deserialization: every distributed communication channel (*e.g.*, ZeroMQ sockets, shared memory ring buffers, PyTorch's TCPStore) uses Python's pickle format to serialize and deserialize data. The pickle protocol allows arbitrary code execution through the `__reduce__()` method, which means any network-reachable attacker can achieve remote code execution by sending a crafted pickle payload (7 CVEs). In addition to that, `vLLM` uses `torch.load()` to load model weight files, which internally uses pickle (2 CVEs). Even after adding the `weights_only=True` safety flag, PyTorch versions before 2.6.0 had a documented bypass [1]

Table 5: Cross-LIF root cause analysis of 60 LIF vulnerabilities.

Root Cause	Sub-Root Causes	CVEs (%)	Classes	LIFs
Unsafe Model File Parsing	Integer overflow, heap buffer overflow, null pointer dereference, division by zero, unbounded allocation	18 (30%)	C1.1, C1.2	llama.cpp (11), Ollama (7)
Untrusted Code Execution	Pickle deserialization, unsafe weight loading, template injection, command injection	12 (20%)	C1.3, C2, C4.7	vLLM (9), LocalAI (2), llama.cpp (1)
Resource Exhaustion	Structured output engine crashes, ReDoS, algorithmic complexity, uncontrolled recursion, unbounded cache growth	12 (20%)	C4.1 - C4.5	vLLM (11), Ollama (1)
Insufficient Access Control	Path traversal, unauthenticated RPC, all-interface binding, SSRF, missing CSRF/origin checks, information disclosure	14 (23%)	C1.4, C3, C4.6, C4.8, C6	Ollama (7), llama.cpp (3), LocalAI (3), vLLM (1)
Total		**60 (100%)**		

that allowed code execution through the restricted unpickler. In `LocalAI` (Go), unsanitized configuration parameters (*e.g.*, backend names, model paths) from user-supplied YAML files are passed directly to process execution, allowing command injection (2 CVEs). In `llama-cpp-python` [19], a Jinja2 chat template embedded in model metadata is rendered without sandboxing, enabling server-side template injection (1 CVE).

Resource Exhaustion / DoS (12 CVEs, 20%) Resource Exhaustion happens when developers provide no limit on user controlled parameters that drive resource consumption. Many of these vulnerabilities reflect developer's false assumptions about input that are reasonable for regular users but become exploitable when an attacker deliberately violates them. This can happen in several forms as described below.

Structured output engines (*e.g.*, `xgrammar`, `outlines`) in `vLLM` crash when processing invalid JSON schemas, regex patterns, or EBNF grammars because these inputs are passed to the C++ compilation backend without pre-validation (3 CVEs). Regular expressions in `vLLM`'s tool call parsers use Python's backtracking regex engine, which allows crafted inputs to cause exponential processing time (2 CVEs). API parameters like `best_of=500` or 500 MB HTTP headers are accepted without limits, exhausting GPU memory or causing server timeouts (3 CVEs). Grammar compilation caches in `xgrammar` [25] and `outlines` [23] grow without size limits or eviction policies, allowing an attacker to exhaust memory by submitting many unique schemas (2 CVEs). In `Ollama`, setting a model path to `/dev/random` causes the server to block forever reading an infinite stream (1 CVE). A separate algorithmic complexity vulnerability in `vLLM`'s multi-modal tokenizer creates a $O(n^2)$ list concatenation on crafted input (1 CVE).

Insufficient Access Control (14 CVEs, 23%) This is the most broadly distributed root cause, affecting all four frameworks we have studied (*i.e.*, `Ollama` has 7, `llama.cpp` 3, `LocalAI` 3, `vLLMS` 1). The core problem here is that LIFs are designed for single-user use on local devices, which, however, are often being deployed in production environment and open to public network. Such a migration implicitly assumes that the default configuration of LIFs that is secure for local use is also safe in networked deployments, which does not align with real-world LIF design. We observe that many components across all four LIFs disable their authentication by default, making LIFs susceptible for unauthorized access in real-world deployment scenarios. Moreover, LIF users often don't check and change the default authentication configuration, resulting in potential security risks.

To be specific, `llama.cpp`'s RPC has no authentication on tensor operations. `Ollama`'s local API misses cross-domain validation. `LocalAI` adopts neither API keys nor cross-site request forgery (CSRF) tokens by default. `vLLM`'s ZeroMQ sockets and PyTorch's TCPStore have no access control, resulting in several security failures. For example, path traversal through model registry workflows in `Ollama` allows attackers to write, delete, or read arbitrary files on the server through crafted digest strings and zip file entries (4 CVEs). Unauthenticated RPC endpoints in `llama.cpp` accept tensor operations from any network client, including a raw memory pointer that enables arbitrary memory read and write (3 CVEs). Internal services in `vLLM` bind to all network interfaces (`0.0.0.0`) regardless of configuration, exposing distributed communication channels to external networks (1 CVE). Server-side request forgery (SSRF) and local file inclusion (LFI) in `LocalAI` allow attackers to scan internal ports and read local files through unvalidated URL parameters (1 CVE). Information disclosure in `Ollama` reflects file system state (*e.g.*, file existence, directory structure) in API error messages (2 CVEs). Web interface vulnerabilities in `LocalAI` exploit the absence of standard web security protections on state-changing endpoints (2 CVEs). DNS rebinding in `Ollama` bypasses the intended localhost-only access control (1 CVE).

5 LIMAScan

To systematically identify LIMA vulnerabilities, we develop LIMAScan, a taxonomy-guided dynamic testing tool that leverages our observed vulnerability patterns, generates crafted payloads for each pattern, and tests those payloads against live LIF instances to trigger crashes or security violations. This section illustrates the design and implementation details of LIMAScan.

5.1 Design

The workflow of LIMAScan is illustrated in Algorithm 1, which takes two inputs (1) a set of vulnerability patterns $\mathcal{P}$, and (2) a set of target LIF instances $\mathcal{F}$, each running inside a Docker container. $\mathcal{P}$ includes the vulnerability patterns identified by our root cause analysis as described in Section 4.2 and Table 5. LIMAScan outputs a set of detected LIMA vulnerabilities $\mathcal{R}$, which records the affected LIF with the corresponding failure and the payload triggering the failure. LIMAScan operates in three phases: malicious payload generation, automated dynamic testing, and crash detection.

Phase 1: Malicious Payload Generation For each vulnerability pattern $p \in \mathcal{P}$, the procedure GENPAYLOAD(p) creates one or more concrete artifacts designed to trigger that specific flaw. LIMAScan generates two types of payloads based on the vulnerability patterns, as described below:

- **Crafted model files** target vulnerabilities in model parsing and loading (corresponding to C1). For example, *integer overflow* (under C1.1) produces three GGUF files, each setting a different header field (key length, array count or value length) to `0x8000000000000000` – a `uint64` value that becomes negative when cast to a signed integer. These file-based payloads can be reused across all LIFs that support the same format: the same crafted GGUF file is tested against `llama.cpp`, `Ollama`, and `LocalAI`, which all support GGUF.
- **Crafted API Requests** target vulnerabilities in runtime input handling (corresponding to C4). For example, the *structured output crashes* pattern (C4.1) sends invalid JSON schemas and regex patterns through the `/v1/chat/completions` endpoint to crash the grammar compilation backend. These payloads are used for LIFs like `vLLM` that do not parse GGUF but expose vulnerable API input processing.

To avoid replicate or reproduce any existing CVE, we generate new payloads based on the pattern instead of reusing the payloads from PoCs. This enables LIMAScan to discover new vulnerabilities that share the similar exploit flow with CVEs in LIFs.

Algorithm 1: LIMAScan

Input: $\mathcal{P} \leftarrow$ vulnerability patterns (from Table 5)
Input: $\mathcal{F} \leftarrow$ LIF instances to test
Output: $\mathcal{R} \leftarrow$ detected vulnerabilities

```
// Phase 1: Generate test payloads from
   patterns
L ← ∅ // initialize
for pattern p ∈ P do
    // crafted GGUF, API request
    L ← L ∪ GENPAYLOAD(p)
// Phase 2: Test payloads against LIFs
R ← ∅
for LIF instance f ∈ F do
    STARTCONTAINER(f)
    WAITUNTILHEALTHY(f)
    for payload l ∈ L applicable to f do
        // copy payload into container
        DEPLOYPAYLOAD(l, f)
        // API call forces LIF to parse l
        SENDREQUEST(f, l)
        // Phase 3: Crash detection
        s ← DETECTCRASH(f)
        if s ≠ NOT_VULNERABLE then
            R ← R ∪ {(f, l, s)}
        if s = CRASHED then
            RESTARTCONTAINER(f)
    STOPCONTAINER(f)
return R
```

DETECTCRASH(f):
Input: $f \leftarrow$ a running LIF instance
Output: $s \leftarrow$ status code

```
// kernel killed for memory
oom ← CHECKOOMKILLED(f)
// abnormal exit
exit ← CHECKEXITCODE(f)
// HTTP health endpoint
health ← CHECKHEALTH(f)
logs ← SEARCHLOGS(f, {panic, SIGSEGV, fatal})
if oom or exit ≠ 0 or ¬health then
    return CRASHED // full denial of service
else if logs ≠ ∅ then
    // crash caught but vuln confirmed
    return PANIC_RECOVERED
else if CHECKINFOLEAK(f) then
    // e.g., env vars in response
    return INFO_LEAK
else
    return NOT_VULNERABLE
```

Table 6: Performance and Precision of evaluating LIMAScan on LIFs.

	Performance		Precision		
			CVEs from LIMABench	Newly Discovered Vulnerabilities	
LIF	**Time (s)**	**#Tests**	**#LIMA / #Others**	**#LIMA / #Others / #Total**	**Patterns Triggering the New Vulnerabilities**
llama.cpp	18	5	0 / 0	0 / 2 / 2	Uncontrolled recursion in grammar/schema parser
vLLM	14	10	0 / 9	0 / 0 / 0	-
Ollama	8	3	1 / 0	2 / 0 / 2	Integer type confusion (uint64 -> int) in GGUF parser
LocalAI	205	15	3 / 0	2 / 1 / 3	Command injection, template injection, unbounded GGUF alloc
Total	**245**	**33**	**4 / 9**	**4 / 3 / 7**	

Phase 2: Automated Dynamic Testing LIMAScan tests each generated payload against every applicable LIF instance. "Applicable" means the payload type matches what the LIF supports; for example, GGUF parsing payloads apply to `llama.cpp`, `Ollama`, and `LocalAI` which all support GGUF, while structured output and ReDoS payloads apply to `vLLM` which exposes grammar compilation and tool call through its API. For each LIF $f \in \mathcal{F}$, the test runner: (1) starts a Docker container with controlled memory limits and a known software version (STARTCONTAINER); (2) waits until the LIF's health endpoint responds (WAITUNTILHEALTHY); (3) iterates over all applicable payloads. For each payload l, the runner delivers the payload to the LIF (DEPLOYPAYLOAD) in two ways: for file-based payloads, this copies the crafted file into the container's model directory; for API-based payloads, the payload content is embedded directly in the request. The runner then sends an API request that forces the LIF to process the payload (SENDREQUEST), and checks whether the LIF crashed (DETECTCRASH). If the LIF crashed, the container is restarted before the test on next payload.

Phase 3: Crash Detection The DETECTCRASH procedure monitors four runtime failure indicators and classifies the outcome into one of the four status codes:

- CRASHED: The LIF process terminated. Detected when: (1) the Docker `OOMKilled` flag is true, meaning the kernel killed the process for exceeding its memory limit; (2) the container exit code is non-zero, which indicates abnormal termination; or (3) the LIF's health endpoint (e.g., `/readyz` for LocalAI, `/health` for llama.cpp) stops responding. A CRASHED result confirms a DoS vulnerability.
- PANIC_RECOVERED: The LIF is still running, but its logs contain crash-related keywords such as `panic`, `SIGSEGV`, or `fatal`. In Go-based LIFs such as `Ollama`, the `recover()` mechanism can catch a panic and keep the process alive, but the presence of a panic confirms the malicious payload has reached and triggered vulnerable code.
- INFORMATION_LEAK: The LIF's API response contains data that should not be accesible. For example, a Sprig [24] template expression such as `{{env "API_KEY"}}` embedded in a model configuration causes `LocalAI` to evaluate the expression and return the server's environment variables in its chat completion response
- NOT_VULNERABLE: None of the above signals were detected; the LIF handled the payload safely.

The completeness of our crash detection depends on whether an LIF internally handles crashes, signals or panics. Our detection cannot capture the failures that are caught and suppressed without logging. However, in our evaluation, none of the evaluated LIFs implement such comprehensive failure handling, allowing our approach to reliably capture crashes in all generated tests.

5.2 Implementation

We implement LIMAScan using Python3, curl and Docker. LIMAScan replies on 18 manually crafted files to support our automated payload generation and testing, which includes: 13 crafted GGUF model files that manipulate header fields to trigger integer overflow, heap overflow, null pointer, and resource exhaustion bugs; 2 structured output files (a GBNF grammar with 50,000 nested parentheses and a JSON schema with 5,000 nested objects); and 3 additional files for path traversal strings, ReDoS patterns, and extreme API parameters. These are delicately designed with specific malicious field values derived from the observed vulnerability patterns.

5.3 Generalized Vulnerability Detection

LIMAScan is primarily designed to detect LIMA vulnerabilities (C1), where a crafted model file exploits weaknesses in a LIF's parsing pipeline. This is the reason why 13 of its 18 payloads are crafted GGUF files. However, its taxonomy-driven design extends naturally to other root causes and vulnerability patterns. Any vulnerabilities triggered by delivering crafted input to a running LIF instance can be used as a pattern, where the input can be model artifacts or user requests. We extend LIMAScan to include the detection of non-LIMA vulnerabilities (C4) summarized in Table 4 by generating API requests with malicious inputs. This extension aims at LIFs without GGUF parsers, such as `vLLM`.

6 Evaluation

We conduct our evaluation of LIMAScan on the most recent leases of the four popular LIFs, including `llama.cpp`, `vLLM`, `Ollama` and `LocalAI`. Here, we report the evaluation procedure and results with our newly discovered vulnerabilities.

Benchmark and Evaluation Methodology We applied LIMAScan to the latest stable releases of the four LIFs by running each LIF inside a Docker container with a 4 GB memory limit: `llama.cpp` build `b8149`, `vLLM` 0.8.3, `Ollama` v0.17.0 and `LocalAI` v3.12.1. Except for `vLLM`, the other three LIFs were evaluated on a Macbook Pro with Apple M1 Po chip (6 Performance Cores and 2 Efficiency Cores, 14-core GPU, 16GB RAM). `vLLM` was evaluated separately on a high computing server with AMD Ryzen Threadripper PRO 7985WX CPU (128 cores, 5.37Ghz), 128GB Ram and a NVIDIA RTX 6000 Ada GPU due to its requirement on GPU access for model loading and inference.

6.1 Performance and Precision of LIMAScan

We report our evaluation results in Table 6 with the execution time, the number of generated automated tests, the number of detected vulnerabilities against the previously reported CVEs classified as LIMA (denoted "#LIMA") and non-LIMA (denoted "#Others") curated by LIMABench. We also present the same metrics for newly discovered vulnerabilities by LIMAScan with their triggering root cause patterns.

In total, LIMAScan generates 33 automated tests for the four containerized LIFs, completing the execution in 245 seconds (61 seconds on average per LIF). LIMAScan detects 13 known CVEs curated in LIMABench, including 4 classified as LIMA and 9 as non-LIMA. This indicates that these vulnerabilities remain unpatched in recent releases despite prior disclosure.

Interestingly, LIMAScan detects and exploits 7 previously unknown vulnerabilities from the most recent stable releases of `llama.cpp`, `Ollama` and `LocalAI`. The vulnerability patterns used to trigger these new security issues are listed in the last column of Table 6, indicating that LIFs continue to exhibit LIMA vulnerabilities even after numerous patches addressing previously disclosed issues.

6.2 New Vulnerabilities Discovered

We designate the seven newly discovered vulnerabilities as *LIMA-NEW-001* through *LIMA-NEW-007*, all of which remain unfixed in the latest stable releases as the time of writing. We have reported all of them to the corresponding LIF GitHub repository maintainers. Here, we present the vulnerabilities in order of severity.

LIMA-NEW-005 is an OS command injection through the Model Context Protocol [20], a standard for connecting language models to external tools through STDIO-based server processes. An attacker who POST to `/models/import` can embed arbitrary shell commands in the `mcpServers` configuration field. These commands are executed via Go's `exec.Command()` without any validation, making it one of the most critical RCE we have found in our evaluation. *LIMA-NEW-006* is a server-side template injection through Go's `text/template` engine with the Sprig [24] function library. A model configuration that contains Sprig expressions like `{{env "HOME"}}` in the chat template field causes the server to evaluate these expressions and leak environment variables. *LIMA-NEW-007* is a DoS crash caused by a crafted GGUF file with `n_kv` set to 4,294,967,295 (*i.e.*, `0xFFFFFFFF`). Then the `gpustack/gguf-parser-go` [18] library reads this value from the header and attempts to allocate memory for over 4 billion KV-pair structures exhausting all available memory and triggering the Linux OOM killer.

LIMAScan found two uncontrolled recursion vulnerabilities in `llama.cpp`, *i.e.*, *LIMA-NEW-003* and *LIMA-NEW-004*. The first one targets the GBNF grammar parser: a grammar string with 50,000 nested parentheses causes mutual recursion between `parse_alternates()` and `parse_sequence()` with no depth limit, overflowing the C++ call stack and crashing `llama-server`. The second targets the JSON schema to grammar converter: a JSON schema with 5,000 nested object properties causes the `visit()` function to call itself recursively without bound, producing the same stack overflow crash.

Two integer type confusion vulnerabilities (*LIMA-NEW-001* and *LIMA-NEW-002*) has been found in `Ollama`. When a crafted GGUF file sets a string length or array element count to `0x8000000000000000`, the `uint64` value converts to a negative `int` in Go. This bypasses the bounds check that compares the length against the scratch buffer size, failing the execution and subsequently crashing the server with a runtime panic.

7 Representative Exploit Case Studies

We select three exploits from our LIMA-NEW discoveries to illustrate how LIMAScan detects vulnerabilities through cross-LIF vulnerability patterns that are summarized by our taxonomy. These cases illustrate three ways that model artifacts can compromise a LIF: (1) by crashing the server through a malformed header value, (2) by killing the server through memory exhaustion, and (3) by achieving arbitrary code execution through a poisoned model configuration.

Model Configuration Achieves RCE in LocalAI (LIMA-NEW-005) `LocalAI` allows users to import model configurations through its `/models/import` API endpoint. In `LocalAI` v3.12.1, the configuration format also supports an `mcp` field for the Model Context Protocol [20]. The

mcp.stdio section contains a mcpServers map where each entry specifies a Command and Args to execute.

The vulnerability is at line 70 from file core/http/endpoints/mcp/tools.go. The function SessionsFromMCPConfig() iterates over the mcpServers entries and passes each server's Command and Args fields directly to Go's exec.Command(): command := exec.Command(server.Command, server.Args...). However, there is no validation, no allowlisting, and no sandboxing for such an important command. The Validate() function in LocalAI's configuration parser only checks that the model.yaml is syntactically correct, which does not inspect the commands that the MCP servers will execute.

An attacker can exploit this vulnerability in two steps. First, the attacker sends a POST request to /models/import with a YAML body that defines a model of which mcp.stdio.mcpServers section contains a crafted entry. For example, the entry can set Command to /bin/sh and Args to ["-c", "touch /tmp/pwned_mcp_rce"]. LocalAI will import this configuration without any warning. Second, the attacker sends a chat completion request to /v1/mcp/chat/completions using the imported model name. Then LocalAI loads the model configuration, calls SessionsFromMCPConfig(), and executes the attacker's command with the same privileges as the LocalAI server. If LocalAI runs as root in the container (which it does by default), the attacker gets root execution inside that container.The command touch /tmp/pwned_mcp_rce will create a file on the server, confirming arbitrary command execution.

This vulnerability was discovered by extending the following pattern we observed across the four LIFs. In llama-cpp-python [19], CVE-2024-34359 [8] achieves RCE through Jinja2 server-side template injection embedded in model metadata. In vLLM, CVE-2025-24357 [28] and CVE-2025-32434 [29] achieve RCE through torch.load() deserializing malicious model weights. In summary, the common pattern here is that model content, whether it is a metadata field, a weight file, or a configuration parameter, can reach a code execution path without sanitization. LIMA-NEW-005 was detected by applying the same pattern that was applied to LocalAI's MCP configuration: the model configuration is treated as data, but it contains executable specifications.

Integer Type Confusion Crashes Ollama (LIMA-NEW-001) When Ollama's GGUF parser reads a string field from a model file, the function readGGUFString() in fs/ggml/gguf.go first reads an 8-byte unsigned integer from the file header to determine the length of the string. It then converts this uint64 value to the Go type int using a direct cast: length := int(llm.ByteOrder.Uint64(buf)). However, the int type in Go is a signed 64-bit type, which means that a uint64 value with its highest bit set, such as 0x8000000000000000, becomes the negative number -9223372036854775808 after the cast. The parser then checks whether this length exceeds the size of a pre-allocated scratch buffer using this condition: if length > len(llm.scratch). Because the length is now negative, and a negative number is always less than the positive buffer size, this bounds check passes. Execution continues to buf = llm.scratch[:length], which attempts to slice the buffer with a negative index. Go does not allow negative slice indices, so the runtime panics with slice bounds out of range [:-9223372036854775808]. The Ollama server processes the crash and must be restarted.

LIMAScan detects this vulnerability using payload negative_key_length.gguf. Our payload generator creates a valid GGUF v3 header with zero tensors and one KV pair. The KV pair's key length field is set to 0x8000000000000000 followed by 16 padding bytes. The entire GGUF file is 48 bytes. LIMAScan's test runner uploads this file to an Ollama v0.17.0 container using the blob API, then calls the model creation endpoint. Ollama's parser reads the file, hits the readGGUFString() function, and crashes. LIMAScan detects this crash by polling the /api/version health endpoint, which does not provide any response.

In addition, we confirmed three variants of this attack by setting the KV key and value string length to 0x8000000000000000, as well as setting the tensor name string length. All three reach the same readGGUFString() function and produce the same panic. The same integer type confusion pattern also appears in readGGUFArray() (LIMA-NEW-002), where a negative array count passed to Go's make() triggers a makeslice: len out of range panic. In llama.cpp, CVE-2025-49847 [31] and CVE-2025-52566 [32] involve similar signed/unsigned integer confusion in the GGUF vocab and tokenizer parsers, where size_t values are narrowed to int32_t. The underlying mistake is the same across both implementations: trusting that a file-supplied integer will fit safely into a signed type. This vulnerability remains unfixed in Ollama v0.17.0. The fix would require validating the uint64 value against a reasonable upper bound before casting it to int.

Unbounded GGUF Allocation Crashes LocalAI (LIMA-NEW-007) LIMAScan's payload generator creates a GGUF file large_n_kv.gguf with the GGUF magic bytes, version 3, zero tensors, and the n_kv field set to 0xFFFFFFFF (4,294,967,295), followed by 64 zero bytes. This file tells any GGUF parser that the model contains over 4 billion KV metadata pairs.

When LIMAScan feeds this file to LocalAI v3.12.1, the request travels through the chat completion API to guessDefaultsFromFile() in core/config/guesser.go, which calls gguf.ParseGGUFFile() from the third-party gpustack/gguf-parser-go [18] library. This library reads the n_kv value from the header and attempts to allocate memory for over 4 billion KV pair structures. The allocation

exhausts all available memory in the container, and the Linux kernel's OOM killer sends SIGKILL to the `LocalAI` process.

This payload was inspired from CVE-2024-23605 [21], a heap buffer overflow in `llama.cpp`'s `gguf_init_from_file_impl()` function caused by the same `n_kv` field. That vulnerability has since been patched in `llama.cpp`, and LIMAScan confirms that `llama.cpp` b8149 handles the payload safely. However, `LocalAI` uses a different GGUF parser written in Go, which was never affected by the original CVE but contains the same missing validation. This result shows that a vulnerability pattern found in one LIF can predict similar bugs in other LIFs, even when the parsers are written in different languages.

8 Discussion and Mitigation

Our analysis reveals that LIMA vulnerabilities concentrate in a small number of root causes. Over half of the vulnerabilities in our dataset (C1 and C2, 32 of 60) trace to trusting model-controlled values in parsing code or using unsafe deserialization. This concentration suggests that targeted hardening of specific components can provide substantial security improvement.

Strict Validation at Parse Boundaries The applied fixes for memory corruption vulnerabilities collected by our dataset were consistently minimal, which often include a single bounds check on a header field or a maximum allocation size limit. Applying the same input validation rigor used in network protocol parsers to model file parsers would prevent most C1 vulnerabilities. The low cost of these fixes suggests that the root cause is the absence of security-oriented development practices for model parsing code rather than technical difficulty.

Safe Serialization Replacing Python's pickle with secure formats that do not support arbitrary code execution, such as MessagePack [46] or Protocol Buffers [49], would eliminate the entire C2 class. The safetensors format implements this principle for weight storage by restricting content to numeric tensors and JSON metadata. Extending this design to interprocess communication would address the systemic unsafe deserialization we observed in distributed inference channels.

Sandboxed Metadata Execution Model metadata fields such as chat templates can include executable content embedded within model files. Rendering such content in sandboxed environments, as demonstrated by the fix for CVE-2024-34359, prevents arbitrary code execution through model metadata and should be applied to related components that interpret model-supplied content during loading.

9 Limitations and Threats to Validity

Our dataset includes only publicly disclosed vulnerabilities indexed in the NVD advisories. Undisclosed or silently patched issues are not captured and may affect prevalence estimates. Although we manually validated each entry against patch commits and reproduction artifacts to ensure correct classification, misinterpretation of advisory descriptions remains possible. Additionally, our analysis focuses on widely adopted LIFs and may not generalize to all emerging inference systems. Finally, our taxonomy reflects recurring patterns observed in current implementations and may evolve as model formats and framework architectures change.

10 Related Work

10.1 Traditional LLM Threats

Prior work on machine learning model and LLM security has focused on detecting malicious model files before loading. Scanning tools such as ModelScan [7] and Hugging Face's built-in scanner [11] inspect serialized files for dangerous patterns, while JFrog [44] measured the prevalence of malicious models on public repositories. Kellas et al. [54] propose per-library deserialization policies that enforce safe loading via pickle. These approaches target the model artifact itself but do not address vulnerabilities in the LIFs that load and process these artifacts.

10.2 Mitigation for LLM Attacks

Sandboxing and trusted execution environments have been explored as containment mechanisms. Several machine learning deployment stacks adopt partial sandboxing through Docker containers [2, 3] or WebAssembly runtimes [16, 53], while system-level approaches restrict system calls [37, 56], file access [65], and network connectivity [68]. Trusted Execution Environments such as AMD SEV and Intel SGX provide stronger isolation [59, 63, 73], but introduce performance overhead [34, 35] and deployment complexity [38, 60]. However, these defenses limit the impact of exploitation without preventing the underlying parsing vulnerabilities. Recent proposals for sandboxing LLM applications [43, 62, 66] remain largely unevaluated in practice.

To our knowledge, no prior work has systematically analyzed and reproduced vulnerabilities in the model loading and initialization pipeline of LIFs. Our work fills this gap by defining the LIMA attack surface, providing a taxonomy, automated reproduction framework and a dynamic testing tool for these vulnerabilities.

11 Conclusion

This paper introduces LIMA, the Local Inference framework-Model Attack surface, and presented a systematic analysis of 60 vulnerabilities across four widely adopted LIFs. Our taxonomy and case studies demonstrate that structured model files serving as a potent attack vector can lead to memory corruption, arbitrary code execution, and credential theft before any inference occurs. The concentration of vulnerabilities in model parsing and unsafe deserialization indicates that a small number of targeted defenses can substantially reduce this attack surface. As LIFs continue to gain popularity and add functionality such as structured output generation, distributed inference, and model registry integration, ongoing security analysis of these expanding code paths will be essential to protect the local and even production-level deployment that users rely on for data privacy.

References

[1] CVE-2025-32434 - GitHub Advisory Database — github.com. https://github.com/advisories/{G}{H}{S}{A}-53q9-r3pm-6pq6. [Accessed 02-03-2026].

[2] Docker Offshore: Smoothing the Waves of Remote-Location MLOps | Docker — docker.com. https://www.docker.com/resources/docker-offshore-smoothing-the-waves-of-remote-location-mlops-dockercon-2023/. [Accessed 08-11-2025].

[3] Full-Stack Reproducibility for AI/ML with Docker & Kaskada | Docker — docker.com. https://www.docker.com/blog/full-stack-reproducibility-for-ai-ml-with-docker-kaskada/. [Accessed 08-11-2025].

[4] ggml-org/llama.cpp: LLM inference in C/C++ — github.com. https://github.com/ggml-org/llama.cpp. [Accessed 16-04-2025].

[5] Hugging Face – The AI community building the future. — huggingface.co. https://huggingface.co. [Accessed 07-11-2025].

[6] Malicious ML models discovered on Hugging Face platform | ReversingLabs — reversinglabs.com. https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face. [Accessed 28-02-2026].

[7] ModelScan | Protect Models From Attacks — protectai.com. https://protectai.com/modelscan. [Accessed 06-11-2025].

[8] NVD - CVE-2024-34359 — nvd.nist.gov. https://nvd.nist.gov/vuln/detail/CVE-2024-34359. [Accessed 07-11-2025].

[9] NVIDIA Triton Inference Server &x2014; NVIDIA Triton Inference Server — docs.nvidia.com. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html. [Accessed 06-11-2025].

[10] OWASP Top 10 for Large Language Model Applications | OWASP Foundation — owasp.org. https://owasp.org/www-project-top-10-for-large-language-model-applications/. [Accessed 22-02-2026].

[11] Pickle Scanning — huggingface.co. https://huggingface.co/docs/hub/en/security-pickle. [Accessed 06-11-2025].

[12] pickle — Python object serialization — docs.python.org. https://docs.python.org/3/library/pickle.html. [Accessed 02-03-2026].

[13] Safetensors — huggingface.co. https://huggingface.co/docs/safetensors/en/index. [Accessed 10-11-2025].

[14] Serving Models | TFX | TensorFlow — tensorflow.org. https://www.tensorflow.org/tfx/guide/serving. [Accessed 06-11-2025].

[15] shubhamsaboo/awesome-llm-apps: Collection of awesome LLM apps with AI Agents and RAG using OpenAI, Anthropic, Gemini and opensource models. - github.com. https://github.com/Shubhamsaboo/awesome-llm-apps. [Accessed 03-04-2025].

[16] WebAssembly and WebGPU enhancements for faster Web AI, part 1 | Blog | Chrome for Developers — developer.chrome.com. https://developer.chrome.com/blog/io24-webassembly-webgpu-1. [Accessed 08-11-2025].

[17] Prompt injection attacks and defenses in llm-integrated applications. *arXiv preprint arXiv:2310.12815*, 2023.

[18] gpustack/gguf-parser-go: GGUF parser for Go. https://github.com/gpustack/gguf-parser-go, 2024.

[19] llama-cpp-python: Python bindings for llama.cpp. https://github.com/abetlen/llama-cpp-python, 2024.

[20] Model context protocol specification. https://modelcontextprotocol.io/specification, 2024.

[21] NVD - CVE-2024-23605. https://nvd.nist.gov/vuln/detail/CVE-2024-23605, 2024.

[22] ollama/ollama: Get up and running with Llama 3, Mistral, Gemma, and other large language models. — github.com. https://github.com/ollama/ollama, 2024. [Accessed 31-05-2024].

[23] Outlines: Structured text generation. https://github.com/dottxt-ai/outlines, 2024.

[24] Sprig: Useful template functions for Go templates. https://github.com/Masterminds/sprig, 2024.

[25] XGrammar: Flexible and efficient structured generation engine for large language models. https://github.com/mlc-ai/xgrammar, 2024.

[26] Here are 18,369 public repositories matching this topic llm — github.com. https://github.com/topics/llm, 2025. [Accessed 25-04-2025].

[27] NVD - CVE-2025-1953. https://nvd.nist.gov/vuln/detail/CVE-2025-1953, 2025.

[28] NVD - CVE-2025-24357. https://nvd.nist.gov/vuln/detail/CVE-2025-24357, 2025.

[29] NVD - CVE-2025-32434. https://nvd.nist.gov/vuln/detail/CVE-2025-32434, 2025.

[30] NVD - CVE-2025-46570. https://nvd.nist.gov/vuln/detail/CVE-2025-46570, 2025.

[31] NVD - CVE-2025-49847. https://nvd.nist.gov/vuln/detail/CVE-2025-49847, 2025.

[32] NVD - CVE-2025-52566. https://nvd.nist.gov/vuln/detail/CVE-2025-52566, 2025.

[33] adia. Data Scientists Targeted by Malicious Hugging Face ML Models with Silent Backdoor — jfrog.com. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/. [Accessed 28-02-2026].

[34] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: towards {High-Performance} serverless computing. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 923–935, 2018.

[35] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. Performance analysis of scientific computing workloads on general purpose tees. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1066–1076. IEEE, 2021.

[36] Daniel Alexander Alber, Zihao Yang, Anton Alyakin, Eunice Yang, Sumedha Rai, Aly A Valliani, Jeff Zhang, Gabriel R Rosenbaum, Ashley K Amend-Thomas, David B Kurland, et al. Medical large language models are vulnerable to data-poisoning attacks. *Nature Medicine*, 31(2):618–626, 2025.

[37] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 212–223, New York, NY, USA, 2010. Association for Computing Machinery.

[38] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[39] Harsh Chaudhari, Giorgio Severi, John Abascal, Matthew Jagielski, Christopher A Choquette-Choo, Milad Nasr, Cristina Nita-Rotaru, and Alina Oprea. Phantom: General trigger attacks on retrieval augmented language generation. 2024.

[40] Zhaorun Chen, Zhen Xiang, Chaowei Xiao, Dawn Song, and Bo Li. Agentpoison: Red-teaming llm agents via poisoning memory or knowledge bases. *Advances in Neural Information Processing Systems*, 37:130185–130213, 2024.

[41] Junjie Chu, Yugeng Liu, Ziqing Yang, Xinyue Shen, Michael Backes, and Yang Zhang. Comprehensive assessment of jailbreak attacks against llms. *arXiv preprint arXiv:2402.05668*, 2024.

[42] Daniela S Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*, pages 275–284. IEEE, 2011.

[43] Zehang Deng, Yongjian Guo, Changzhou Han, Wanlun Ma, Junwu Xiong, Sheng Wen, and Yang Xiang. Ai agents under threat: A survey of key security challenges and future pathways. *ACM Computing Surveys*, 57(7):1–36, 2025.

[44] drewt. JFrog and Hugging Face Join Forces to Expose Malicious ML Models — jfrog.com. https://jfrog.com/blog/jfrog-and-hugging-face-join-forces. [Accessed 06-11-2025].

[45] Mohamed Amine Ferrag, Norbert Tihanyi, Djallel Hamouda, Leandros Maglaras, and Merouane Debbah.

From prompt injections to protocol exploits: Threats in llm-powered ai agents workflows. *arXiv preprint arXiv:2506.23260*, 2025.

[46] Sadayuki Furuhashi. MessagePack: It's like JSON, but fast and small. https://msgpack.org/, 2008. Accessed: 2026-03-03.

[47] Julia Gomez-Rangel, Young Lee, and Bozhen Liu. Security in the wild: An empirical analysis of llm-powered applications and local inference frameworks. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, pages 149–159. IEEE, 2025.

[48] Julia Gomez-Rangel, Alvaro Vazquez, Young Lee, Kadir Alpaslan Demir, and Bozhen Liu. Rising fast, prone to risk: How open-source llm-powered apps are designed and secured. In *Proceedings of the 2025 International Workshop on Artificial Intelligence× Software Engineering (AIxSE). IEEE*, 2025.

[49] Google. Protocol Buffers: Google's data interchange format. https://protobuf.dev/, 2008. Accessed: 2026-03-03.

[50] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pages 79–90, 2023.

[51] W Ronny Huang, Jonas Geiping, Liam Fowl, Gavin Taylor, and Tom Goldstein. Metapoison: Practical general-purpose clean-label data poisoning. *Advances in Neural Information Processing Systems*, 33:12080–12091, 2020.

[52] Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M Ziegler, Tim Maxwell, Newton Cheng, et al. Sleeper agents: Training deceptive llms that persist through safety training. *arXiv preprint arXiv:2401.05566*, 2024.

[53] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. Wave: a verifiably secure webassembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2940–2955. IEEE, 2023.

[54] Andreas D Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P Kemerlis, James C Davis, and Junfeng Yang. Pickleball: Secure deserialization of pickle-based machine learning models. *arXiv preprint arXiv:2508.15987*, 2025.

[55] Keita Kurita, Paul Michel, and Graham Neubig. Weight poisoning attacks on pretrained models. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 2793–2806, 2020.

[56] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster system calls through sandboxed privileged execution. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[57] Linyang Li, Demin Song, Xiaonan Li, Jiehang Zeng, Ruotian Ma, and Xipeng Qiu. Backdoor attacks on pre-trained models by layerwise weight poisoning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3023–3032, 2021.

[58] Yanzhou Li, Tianlin Li, Kangjie Chen, Jian Zhang, Shangqing Liu, Wenhan Wang, Tianwei Zhang, and Yang Liu. Badedit: Backdooring large language models by model editing. *arXiv preprint arXiv:2403.13355*, 2024.

[59] Junming Ma, Chaofan Yu, Aihui Zhou, Bingzhe Wu, Xibin Wu, Xingyu Chen, Xiangqun Chen, Lei Wang, and Donggang Cao. S3ml: A secure serving system for machine learning inference. *arXiv preprint arXiv:2010.06212*, 2020.

[60] Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively.* PhD thesis, Carnegie Mellon University, USA, 2016.

[61] Ian R McKenzie, Oskar J Hollinsworth, Tom Tseng, Xander Davies, Stephen Casper, Aaron D Tucker, Robert Kirk, and Adam Gleave. Stack: Adversarial attacks on llm safeguard pipelines. *arXiv preprint arXiv:2506.24068*, 2025.

[62] James Mickens, Sarah Radway, and Ravi Netravali. Guillotine: Hypervisors for isolating malicious ais. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 18–26, New York, NY, USA, 2025. Association for Computing Machinery.

[63] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. Machine learning with confidential computing: A systematization of knowledge. *ACM Comput. Surv.*, 56(11), June 2024.

[64] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[65] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, 2001.

[66] Shenao Wang, Yanjie Zhao, Zhao Liu, Quanchen Zou, and Haoyu Wang. Sok: Understanding vulnerabilities in the large language model supply chain. *arXiv preprint arXiv:2502.12497*, 2025.

[67] Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? *Advances in Neural Information Processing Systems*, 36, 2024.

[68] Jay Weinstein, Mark Fenkner, Charles King, Ismael Lopez, and Peter Martz. Sandbox based internet isolation in an untrusted network, February 4 2020. US Patent 10,554,475.

[69] Baoyuan Wu, Hongrui Chen, Mingda Zhang, Zihao Zhu, Shaokui Wei, Danni Yuan, and Chao Shen. Backdoorbench: A comprehensive benchmark of backdoor learning. *Advances in Neural Information Processing Systems*, 35:10546–10559, 2022.

[70] Zhen Xiang, Fengqing Jiang, Zidi Xiong, Bhaskar Ramasubramanian, Radha Poovendran, and Bo Li. Badchain: Backdoor chain-of-thought prompting for large language models. *arXiv preprint arXiv:2401.12242*, 2024.

[71] Jiashu Xu, Mingyu Ma, Fei Wang, Chaowei Xiao, and Muhao Chen. Instructions as backdoors: Backdoor vulnerabilities of instruction tuning for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3111–3126, 2024.

[72] Jingwei Yi, Yueqi Xie, Bin Zhu, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. Benchmarking and defending against indirect prompt injection attacks on large language models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, pages 1809–1820, 2025.

[73] Qihang Zhou, Wenzhuo Cao, Xiaoqi Jia, Peng Liu, Shengzhi Zhang, Jiayun Chen, Shaowen Xu, and Zhenyu Song. Rcontainer: A secure container architecture through extending arm cca hardware primitives. NDSS, 2025.

[74] Andy Zou, Zifan Wang, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

[75] Wei Zou, Runpeng Geng, Binghui Wang, and Jinyuan Jia. {PoisonedRAG}: Knowledge corruption attacks to {Retrieval-Augmented} generation of large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3827–3844, 2025.