Secure-Torch: A Defense-In-Depth Trust Gateway for AI Model Deserialization

Atharva Khambete[1], Avanish Kulkarni[2]
[1,2]*Student, Information Technology, Marathwada Mitra Mandal's College of Engineering, India*

***Abstract*—These instructions give you guidelines for preparing papers for the International Journal of Innovative Research in Technology (IJIRT). Model files have become an overlooked software supply-chain boundary. Common machine learning loading paths deserialize artifacts with limited trust controls, enabling attack classes such as pickle gadget execution, metadata-triggered payloads, and unsafe custom operator loading. We present secure-torch, a Python library that inserts a fail-closed trust enforcement layer into model loading workflows. The system provides a fixed-order six-stage pipeline: format detection, signature verification, threat scoring, policy enforcement, optional sandbox isolation, and controlled load return. The implementation supports PyTorch pickle-based artifacts, SafeTensors, and ONNX, combining static artifact inspection with provenance and policy gates. Evaluation against known common vulnerabilities and exposures (CVEs) indicates strong baseline coverage for deserialization attack patterns and practical compatibility with incremental adoption via audit mode. We discuss the architecture, implementation trade-offs, and limitations relevant to advancing secure ML deployment research.**

***Index Terms*—Model supply chain security, provenance verification, PyTorch security, safe deserialization, sandboxing, threat scoring.**

I. INTRODUCTION

The machine learning (ML) ecosystem increasingly consumes third-party model artifacts from public repositories, model hubs, and internal registries. In most deployments, loading is treated as a utility operation rather than a security boundary. This assumption is inherently risky because model artifacts may embed arbitrary behavior that executes during deserialization or model initialization. Recent vulnerability classes and advisories surrounding pickle-based loading and ONNX custom operators reinforce this critical operational concern.[7]

The secure-torch project frames model loading fundamentally as a trust gateway problem. Instead of relying on a single, isolated mitigation strategy, it layers format-aware inspection, provenance checks, policy evaluations, and process isolation. The primary design goal is to ensure malicious or unverifiable artifacts fail closed by default, while preserving practical API compatibility with existing PyTorch-centered workflows.[4]

A. Contributions

This work makes the following contributions to ML supply chain security:

- A unified secure loading API supporting PyTorch-compatible usage patterns with injected security controls.
- A fixed-order validation pipeline that prevents ad hoc bypass of critical security checks.
- Format-specific static validators for pickle streams, SafeTensors metadata, and ONNX graph domains.
- Dual provenance modes: online Sigstore-style verification and offline public-key signature verification.
- Explainable additive threat scoring linked to threshold-based policy enforcement.
- Optional sandboxed load paths utilizing subprocess isolation and Linux seccomp hardening.

II. RELATED WORK

The intersection of machine learning operations (MLOps) and cybersecurity has seen rapid development as the ecosystem increasingly relies on third-party model sharing. The challenge of safe model serialization and deserialization has prompted several targeted solutions, which can be broadly categorized into format innovations, static analysis tools, and

supply chain provenance frameworks.[3]

A. Vulnerabilities in ML Serialization

The standard serialization format in the Python ecosystem, pickle, is inherently insecure because it allows arbitrary code execution during the deserialization phase via the reduce method and specific opcodes like GLOBAL and INST. Recent advisories have demonstrated how these legacy formats can be weaponized to achieve remote code execution (RCE) upon simply loading a PyTorch model. Similarly, complex formats like ONNX introduce risk through custom operator domains and external data references, which can be manipulated for path traversal or malicious execution.[11]

B. Format-Specific Mitigations

To address the structural flaws of pickle, Hugging Face introduced SafeTensors, a zero-copy for-mat specifically designed to store tensors without allowing executable code, effectively mitigating traditional de-serialization payloads. Concurrently, PyTorch has implemented the weights_only=True parameter in its core torch.load() function, which restricts the unpickler to a strict whitelist of known safe classes and tensor types. While these format-level restrictions are highly effective, they require ecosystem-wide migration or explicit developer opt-in, leaving legacy models vulnerable.[12]

C. Static Analysis and Scanning Ecosystems

In response to the proliferation of untrusted models, tools such as picklescan and Model Scan were developed to perform static analysis on model files. These tools parse the opcode streams of legacy pickle files and the metadata of newer formats to detect known malicious patterns without executing the payload. Major model hubs have integrated similar scanning mechanisms into their ingestion pipelines. However, these scanners typically operate as out-of-band checks during storage or download, rather than inline enforcement gateways during the actual runtime load operation.[13]

D. Supply Chain Provenance

Broader software supply chain security frameworks are actively being adapted for ML artifacts. Technologies such as Sigstore allow for cryptographic signing and verification of models, while Software Bill of Materials (SBOM) standards, like SPDX, provide transparency into a model's components and lineage. Frameworks like in-toto help cryptographically attest to the steps taken in the ML supply chain, ensuring that the model has not been tampered with between training and deployment.

E. The Need for Compositional Defense

While the aforementioned tools provide strong localized mitigations, they often lack a unified, compositional defense strategy. Static scanners can be bypassed by obfuscation, format restrictions do not prevent supply chain spoofing, and provenance checks do not analyze the actual payload. secure-torch addresses this gap by integrating format-aware static analysis, cryptographic provenance verification, Open Policy Agent (OPA) evaluations, and OS-level sandboxing (seccomp) into a single, cohesive, fail-closed enforcement layer that executes directly at the runtime load boundary.

III. SYSTEM OVERVIEW

A. Architecture

The secure-torch architecture is positioned as an intermediary between model distribution sources (e.g., local disk, remote hubs) and the runtime ML loaders. As illustrated in Fig. 1, by intercepting the load call, the system forces the artifact through a rigid validation protocol before memory allocation occurs.[10]

B. Fixed Pipeline

To prevent bypass attacks, the secure load path is intention-ally deterministic and ordered. As shown in Fig. 2, it executes the following sequential stages:

1) Format Detection: Analyzes file extensions and magic bytes.

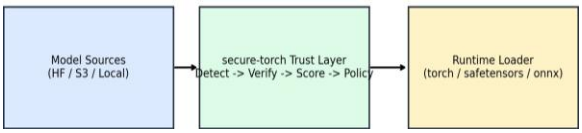

Fig. 1. Position of secure-torch in model artifact ingestion.

2) Provenance Verification: Validates cryptographic sig-natures.
3) Threat Scoring: Executes format-specific static

validators.

4) Policy Evaluation: Assesses SBOMs and user-defined trust policies.
5) Execution Strategy: Dispatches to an isolated sandbox or direct load based on configuration.
6) Return/Report: Yields the model or an audit report if the threat threshold is exceeded.

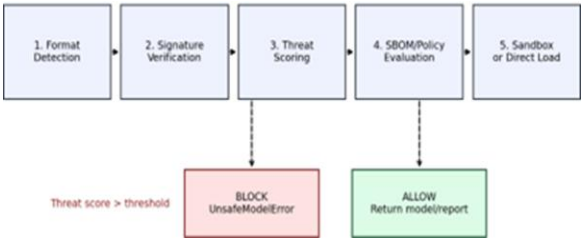

Fig. 2. Secure load pipeline and block/allow decision gate.

IV. IMPLEMENTATION DETAILS

A. Core Components

Table I summarizes the principal architectural components of the proposed system. Furthermore, Fig. 3 illustrates the codebase distribution, highlighting that the highest code complexity resides in the load orchestration and sandboxing mechanisms.[8]

B. Threat Scoring Model

Threat scoring is designed to be additive and transparent. Let the total threat score be defined as:

$$S = \sum_{i=1}^{n} s_i \quad (1)$$

where each s_i represents a named contribution from a specific validator or policy check. The enforcement mechanism blocks artifact loading when $S > \tau$, with the default threshold currently established at $\tau = 20$. Selected score constants are outlined in Table II and visualized in Fig. 4.

Table I Architectural Components of The Trust Gateway

Logical Component	Responsibility
Load Orchestrator	Orchestrates the secure pipeline and execution dispatch.
Format Detector	Detects artifact format via magic bytes.
Pickle Static Analyzer	Walks pickle opcodes without executing arbitrary payloads.
SafeTensors Validator	Validates headers, metadata, and dtypes for code injection.
ONNX Graph Inspector	Inspects ONNX domains, nested graphs, and external data.
Provenance Verifier	Handles online/offline signature verification (Sigstore).
Trust Enforcer	Matches publishers against defined trust policies.
Policy Evaluator	Parses SPDX and runs Open Policy Agent (OPA) rules.
Subprocess Sandbox	Manages isolated subprocess loads and transfer protocols.
Seccomp Hardening	Enforces OS-level syscall filtering for Linux environments.
Hub Interceptor	Optionally patches standard hub download paths for scanning.

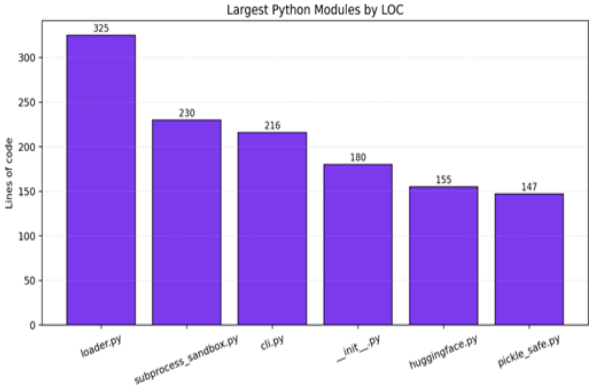

Fig. 3. Largest Python modules by Lines of Code (LOC) in the repository.

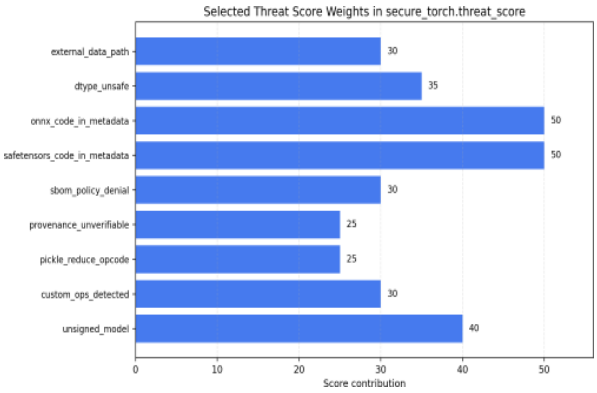

Fig. 4. Selected threat score weights applied during heuristic evaluation.

Table II Selected Threat Score Weights for Heuristic Evaluation

Detected Anomaly / Threat Indicator	Score
Unsigned model artifact	40
Custom operators detected (ONNX)	30
Pickle REDUCE opcode utilized	25
Pickle global unknown module referenced	10
Pickle INST opcode utilized	10
Executable code detected in SafeTensors metadata	50
Executable code detected in ONNX metadata	50
Suspicious ONNX external data path	30
Provenance signature unverifiable	25
SBOM policy denial (OPA rejection)	30

C. Sandboxing Strategy

The sandbox mode executes the deserialization sequence within a constrained subprocess to significantly reduce the blast radius of a successful exploit. The implementation strips proxy and network-related environment variables in the child process, utilizes an inter-process JSON transfer protocol rather than standard unpickling, and applies strict Linux seccomp filtering where supported by the host operating system.

V. SECURITY EFFICACY VALIDATION

A. Detection Coverage

To evaluate the efficacy of the proposed gateway, the implementation was subjected to a series of targeted vulnerability tests representing known attack vectors in the ML ecosystem. Figure 5 summarizes the robust passing rate of the local test suite validating these defenses. The static analysis modules successfully demonstrated coverage against critical threat models, including:

- Pickle Gadget Exploits: Reliable interception of GLOBAL, STACK_GLOBAL, REDUCE, and INST opcode sequences often used for arbitrary code execution.
- Dangerous Module Imports: Blocking of unsafe callable references such as os, subprocess, builtins.eval, and ctypes.
- Metadata Injection: Identification of obfuscated payload patterns within SafeTensors headers.
- Graph Manipulation: Detection of unverified ONNX custom operator domains and path-traversal attempts in external data pointers.

B. Vulnerability Regression Validation

The framework was evaluated against specific common vulnerabilities and exposures (CVEs) related to model deserialization (e.g., CVE-2023-44271, CVE-2024-5980). The pipeline consistently flagged the malicious artifacts, proving that while static analysis cannot formally guarantee the absence of zero-day exploits, it provides robust guardrails against previously weaponized patterns.

C. Operational Integration

To ensure practical adoption without disrupting engineering workflows, the tool implements an audit only=True

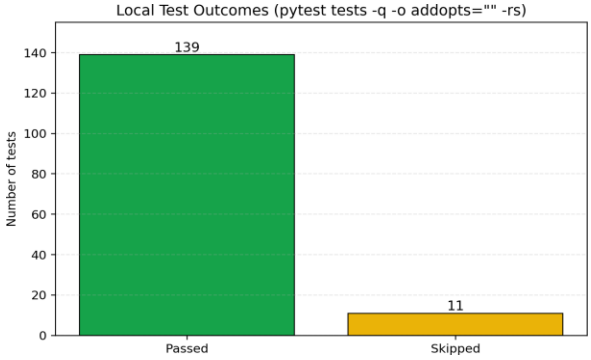

Fig. 5. Local test suite outcomes (pytest execution summary).

parameter. This allows teams to deploy the gateway in a non-blocking mode, gathering telemetry on threat scores before enforcing hard thresholds in production environments.

VI. CONCLUSION AND FUTURE SCOPE

Secure-torch demonstrates that model loading must be treated as a rigorous security boundary. Its fixed-order pipeline, explainable threat scoring, and optional sandboxing provide a reusable defense-in-depth baseline for artifact ingestion. The current architecture proves viable against major deserialization attack classes. Future work will focus on extensive performance benchmarking to quantify the latency overhead of the subprocess sandbox on enterprise-scale models (e.g., LLMs exceeding 70B parameters) and the integration of dynamic, memory-safe Rust extensions for faster format inspection.

REFERENCES

[1] NIST, "Secure Software Development Framework (SSDF)," NIST SP 800-218, 2022.
[2] Linux Foundation, "SPDX Specification," [Online]. Available: https://spdx.dev
[3] in-toto Project, "in-toto: A framework to secure the software supply chain," [Online]. Available: https://in-toto.io
[4] Python Software Foundation, "pickle — Python object serialization," [Online]. Available: https://docs.python.org/3/library/pickle.html
[5] PyTorch, "torch.load Documentation," [Online]. Available: https://pytorch.org/docs/stable/generated/torch.load.html
[6] Hugging Face, "SafeTensors: Safe and fast tensor serialization," [Online]. Available: https://github.com/huggingface/safetensors
[7] ONNX, "Open Neural Network Exchange (ONNX)," [Online]. Available: https://onnx.ai
[8] Linux Kernel Organization, "seccomp (Secure Computing Mode)," [Online]. Available: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
[9] OWASP Foundation, "OWASP Top 10: Software and Data Integrity Failures," [Online]. Available: https://owasp.org/Top10/
[10] Open Policy Agent, "OPA Documentation," [Online]. Available: https://www.openpolicyagent.org
[11] Sigstore Project, "Sigstore: Software Supply Chain Security," [Online]. Available: https://sigstore.dev