An Empirical Study on Remote Code Execution in Machine Learning Model Hosting Ecosystems

Anonymous Author(s)

Abstract

Model-sharing platforms such as Hugging Face, ModelScope, and OpenCSG have become central to modern machine learning development, enabling researchers and practitioners to share, load, and fine-tune pre-trained models with minimal effort. However, the flexibility of these ecosystems introduces a critical security concern: the execution of untrusted code during model loading (*e.g.*, via `trust_remote_code` or `trust_repo`). In this work, we conduct the first large-scale empirical study of custom model loading practices across five major model-sharing platforms to assess their prevalence, associated risks, and developer perceptions. We first quantify the frequency with which models require custom code to function and identify those that execute arbitrary Python files during loading. We then apply three complementary static analysis tools—Bandit, CodeQL, and Semgrep—to detect security smells and potential vulnerabilities, categorizing our findings by CWE identifiers to provide a standardized risk taxonomy. In parallel, we systematically analyze the documentation, API design, and safety mechanisms of each platform to understand their mitigation strategies and enforcement levels. Finally, we conduct a qualitative analysis of over 600 developer discussions from GitHub, Hugging Face, and PyTorch Hub forums, and Stack Overflow to capture community concerns and misconceptions regarding security and usability. Our findings reveal widespread reliance on unsafe defaults, uneven security enforcement across platforms, and persistent confusion among developers about the implications of executing remote code.

CCS Concepts

• **Security and privacy** → **Systems security**; **Software security engineering**; • **Software and its engineering** → *Software infrastructure*.

Keywords

Large Language Models (LLMs), Software Security, Remote Code Execution, Model Hub

ACM Reference Format:

Anonymous Author(s). 2026. An Empirical Study on Remote Code Execution in Machine Learning Model Hosting Ecosystems. In *Proceedings of The Mining Software Repositories (MSR), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference'17, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

In recent years, Large Language Models (LLMs) such as ChatGPT and Gemini have made remarkable progress in day-to-day conversation and assisting tasks. These models are based on different transformer architectures and their advancements. These have enabled the creation of models with unprecedented scale, often comprising billions or even trillions of parameters. Models are continuously reused, re-tuned, and evaluated for new tasks. Model hubs (or model registries) like Hugging Face play a critical role in this ecosystem by providing a centralized platform for hosting and sharing pre-trained models and datasets [49]. As of October 2025, Hugging Face alone hosts over 1.7 million models, fostering an open and collaborative environment for developers and researchers worldwide.

While model hubs and their supporting libraries, such as transformers and PyTorch, enable the seamless distribution of model weights, some models inherently require code execution to function correctly [16]. Early neural network architectures relied on standardized, composable layers that could be fully described through configuration files. In contrast, current large language models often introduce non-standard architectural components—including custom attention mechanisms, domain-specific preprocessing steps, and hardware-aware optimizations—that cannot be represented without executable code [48]. For example, when researchers develop a new transformer variant with a novel positional encoding scheme, they must distribute not only the trained weights but also the accompanying Python code that specifies how those weights are applied during inference. Without this code, downstream users would need to manually re-implement the architecture, making model sharing inefficient and, in many cases, impractical [7].

The requirement for code execution during model loading offers significant flexibility and extensibility, but it also introduces critical security risks. A key example is the widespread use of Python's pickle protocol for model serialization, which allows arbitrary code execution during deserialization through the __reduce__ method—effectively rendering data files executable payloads rather than passive objects [23]. Zhao et al. identified 91 malicious models in the wild that exploited these attack vectors to deploy reverse shells, steal credentials, and install cryptominers [49]. Model loading from the hubs establishes transitive trust relationships that users often overlook. By loading a model, users implicitly place trust not only in the model's author but also in the platform's security scanning, the framework's isolation guarantees, the integrity of upstream dependencies, and the safety of the local runtime environment. In practice, many users may equate high download counts with

safety, assume that verified organizations imply trustworthy content, and rely on platforms to prevent the distribution of malicious artifacts—assumptions that have repeatedly proven to be unreliable [21].

In our work, we focus on systematically analyzing the security risks and ecosystem implications of untrusted model code execution on model-sharing platforms. Although model hubs play a crucial role in providing access to AI models, the ability to execute arbitrary code during model loading introduces a powerful yet poorly understood attack surface. Our study aims to close this gap by examining the prevalence of custom model loading, identifying its security risks, evaluating platform-level mitigation strategies, and characterizing developer perceptions through discussions across different platforms.

Our contributions are:

- The first cross-platform, large-scale measurement study of untrusted model code execution across five major model-sharing platforms: Hugging Face, ModelScope, OpenCSG, OpenMMLab, and PyTorch Hub.
- Systematically detect and categorize security weaknesses using three static analysis tools—Bandit, CodeQL, and Semgrep in around 45,000 repositories containing custom code.
- Analyze platform-level defenses, including warning systems, static and dynamic scanning, and trust flag mechanisms.
- A taxonomy about developers' perception about remote code execution during model loading after examining around 600 developer discussions from forums, GitHub issues, pull requests, and Q&A sites.

2 Background

This section establishes the fundamental concepts necessary to understand the work.

2.1 Code Execution in Model Sharing Ecosystems

Unlike traditional data files, modern machine learning models may require code execution for fundamental technical reasons. While early neural networks consisted of standardized layers that could be described solely by configuration, contemporary deep learning architectures implement novel mechanisms – custom attention patterns, domain-specific preprocessing, and hardware optimizations – that cannot be expressed without executable code [48]. When researchers develop a new transformer variant with unique positional encoding, they must ship both the trained weights and the Python code defining how those weights interact. The alternative would require every user to manually reconstruct the architecture, making model sharing impractical [7].

This technical necessity manifests through platform APIs that appear deceptively simple. When users call `from_pretrained` methods, the framework downloads multiple files, including Python modules that execute with full system privileges when remote code trust flags are enabled. Platforms like Hugging Face host over 1.7 million models, with thousands added daily, making manual code review impossible and automated scanning essential yet insufficient [24].

2.2 Code Quality and Security Smells

Code quality broadly refers to the degree to which source code is reliable, maintainable, and free from defects [22]. It is commonly assessed through indicators such as *defect rate* (e.g., number of defects per line of code or function point) and *reliability* (e.g., mean time to failure or number of failures over time). High-quality code typically adheres to standardized coding practices and design principles, which promote readability, consistency, and long-term maintainability.

Code smells are indicators of poor design or implementation choices that may not immediately cause failures but often lead to maintainability issues and increased defect risk [4, 39]. They typically reflect violations of good design principles, making software harder to evolve and more error-prone. For example, in Listing 1 (left), the ValueError exception is caught but not handled—an anti-pattern that reduces error transparency and can lead to subtle failures [6].

Listing 1 Examples of a code smell (left) and a security smell (right).

Code smell example

```
try:
    num = input('Enter number:')
    num = int(num)
except ValueError:
    pass
```

Security smell example

```
import hashlib
def validate(c, h):
    hash_md5 = hashlib.md5(c)
    hash = hash_md5.hexdigest()
    return hash == h
```

A specific subset of code smells—***security smells***—is associated with patterns that may introduce or signal the presence of vulnerabilities [5, 45, 46]. These patterns do not always constitute direct exploits but highlight code areas where security controls are weak or outdated. For example, in Listing 1 (right), the use of the insecure `md5` algorithm is associated with *CWE-327: Use of a Broken or Risky Cryptographic Algorithm* [47]. Such patterns increase the likelihood of exploitation if left unaddressed.

3 Threat Model

Custom model loading introduces complex trust boundaries among three main stakeholders: *model creators*, *platform maintainers*, and *model consumers*. Our threat model examines how these boundaries can be exploited when platforms allow arbitrary code execution during model loading.

Adversary Assumptions. We assume that attackers have access to public-facing interfaces of model-sharing platforms and can upload models or modify existing repositories. Adversaries may also interact with community features such as discussions or pull requests to promote malicious content. We consider two primary adversary types: (1) a *malicious contributor* who intentionally uploads a model with harmful initialization code, and (2) a *compromised maintainer or trusted account* whose credentials or tokens are hijacked to distribute poisoned models under the guise of a reputable author.

Trust Relationships. Model consumers implicitly trust platform interfaces and configuration defaults (e.g., `trust_remote_code=True` or `trust_repo=True`) to safely retrieve and execute model code. This trust is often amplified by perceived platform reputation or

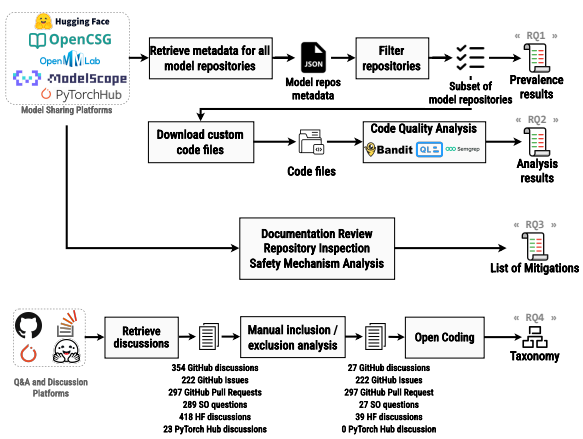

Figure 1: Methodology

download counts, which may lead users to overlook warning prompts or disable security mechanisms for convenience. Platform maintainers, in turn, trust model contributors to follow safe publishing practices, while contributors depend on the platform to enforce isolation and verification mechanisms. The intersection of these assumptions creates a vulnerable trust boundary.

Attack Surface. An attacker can exploit these trust relationships in several ways:

- **Malicious Initialization Code.** Injecting arbitrary Python logic in files such as `modeling_*.py`, `tokenizer.py`, or `hubconf.py` that execute automatically when users load models with trust flags enabled.
- **Dependency and Supply Chain Manipulation.** Leveraging insecure imports or dependency confusion to pull malicious packages during model loading or installation.
- **Platform-level Exploits.** Abusing insufficient sandboxing or lack of static checks to execute code within the environment of the end-user or cloud-hosted inference services.
- **Social Engineering and Trust Hijacking.** Using reputation metrics (e.g., verified badges, download counts) or community interactions to mislead users into enabling unsafe options.

Scope. Our threat model does not assume insider access to platform infrastructure or zero-day vulnerabilities in hosting environments. Instead, it focuses on adversaries exploiting *design-level weaknesses*—specifically, unsafe defaults, inadequate code vetting, and ambiguous developer guidance that blur the line between flexibility and security.

4 Methodology

4.1 Research Questions

In this work, we are going to answer four research questions that explore the prevalence, risks, and developer perceptions of custom model loading across model sharing platforms.

RQ1: To what extent is custom model loading required?

Model-sharing platforms, such as Hugging Face and ModelScope, enable developers to provide custom code for loading or configuring models. While these capabilities increase flexibility and support novel architectures, they also introduce security risks within the ecosystem. In this RQ, we investigate how many models hosted on these platforms include custom model loading code.

RQ2: Are the remote codes with custom models vulnerable and security smelly?

Allowing the execution of arbitrary code raises concerns about the introduction of vulnerabilities and insecure practices. Previous work has shown that code provided by the community may contain security smells, such as unsafe deserialization [49]. In this research question, we perform a systematic analysis of the code associated with the custom model to determine the prevalence of vulnerabilities and potential exploit vectors.

RQ3: What does the platform offer for developers to mitigate the code execution during model loading?

Platform maintainers play a crucial role in enforcing safe defaults, providing comprehensive documentation, and implementing technical safeguards (*e.g.*, sandboxing, warning banners, or permission systems). In this research question, we investigate the existing security mechanisms provided by various platforms. We examine whether platforms provide static or dynamic checks, whether they expose developers and end-users to explicit warnings when running custom code, and how policies such as `trust_remote_code` or `trust_repo` flags or isolated execution environments are enforced in practice.

RQ4: What are the developers' concerns around code execution during model loading?

Beyond technical vulnerabilities, it is crucial to understand the perspective of developers who contribute to and consume these models. Their concerns may range from usability (*e.g.*, friction in using security mechanisms) to trustworthiness (*e.g.*, fear of executing malicious code) and maintainability (*e.g.*, lack of long-term platform support for their contributions). In this research question, we collect and analyze developer discussions from Hugging Face and PyTorch Hub forums, GitHub discussions, pull requests, issues, and StackOverflow Q&A platforms to understand the concerns, misconceptions, and expectations of practitioners.

4.2 Platform Selection

First, we examined a recent list of popular model-sharing platforms compiled by Jian *et al.* [49], which identified the top fifteen model hubs. We then applied the following inclusion criteria to determine whether a given model hub (henceforth referred to as a "platform") qualified for our study. A platform was included only if it satisfied *all* of the following conditions:

- The platform is publicly accessible.

- The platform provides a practical means to retrieve information from all hosted model repositories.
- The hosted models can be directly loaded using existing libraries such as `transformers`, `torch.hub`, or their respective wrappers, rather than requiring custom scripts to load models from scratch.
- The platform supports libraries or mechanisms that permit the execution of custom code during model loading.

After this manual analysis, we identified 5 platforms to be included in our study, listed in Table 1.

Table 1: Top 15 Model Hubs.

Model Hub	#Models Repo	Included
Hugging Face	1,699,968	✓
SparkNLP	-	✗
OpenCSG	192,556	✓
Kaggle	-	✗
ModelScope	68,736	✓
ModelZoo	-	✗
OpenMMLab	16	✓
ONNX Model Zoo	-	✗
NVIDIA NGC	-	✗
MindSpore	-	✗
WiseModel	-	✗
PaddlePaddle	-	✗
SwanHub	-	✗
Liandanxia	-	✗
PyTorch Hub	26	✓

4.3 Custom Model Repositories Collection

After selecting the platforms, we collected metadata for all model repositories, specifically including the repository's link, tags, and file lists. The numbers of collected repositories are presented in Table 1. Then, we checked one of the following conditions to meet to include a repository that needs custom code executions:

- If the repository was tagged as `custom_code`.
- There was one of the following files: `tokenizer.py`, `__init__.py`, or `hubconf.py`.
- There was one of the Python files that started with: `modeling_`, `tokenization_`, `configuration_`.

These conditions are adapted from `transformers` [16] and `Pytorch Hub` [44] documentation. The OpenMMLab platform has its own library for loading custom models; therefore, we included all the repositories listed on the platform in our analysis.

4.4 Security Smell Analysis

After collecting the models, we used three static analyzers to identify security smells and potential vulnerabilities: Bandit, CodeQL, and Semgrep. These tools complement each other in terms of analysis depth and coverage.

Bandit. Bandit is a Python-specific security linter that statically inspects the abstract syntax tree (AST) of source code to detect common vulnerabilities such as the use of unsafe functions (`eval`, `exec`, `pickle.load`), weak cryptographic algorithms, hardcoded credentials, and insecure temporary file creation. We executed Bandit version 1.8.6 recursively on all Python files extracted from each model repository, generating structured `JSON` outputs for aggregation and comparison. Bandit also maps findings to *Common Weakness Enumeration (CWE).*

CodeQL. CodeQL performs static analysis by compiling source code into a relational database of program elements (e.g., functions, variables, control flow, and data flow) and executing declarative `QL` queries to detect security flaws. It enables inter-procedural and data-flow analysis for complex vulnerabilities such as injection, path traversal, and insecure deserialization. For Python-based repositories, we used version 2.15.0 and the official query pack provided by GitHub Security Lab.

Semgrep. Semgrep is a lightweight, multi-language static analyzer that uses rule-based pattern matching to detect both general and domain-specific security issues. Unlike CodeQL, which requires query compilation, Semgrep matches syntactic and semantic patterns directly in the codebase, making it efficient for large-scale scanning. We employed version 1.139.0, which includes built-in rulesets, to identify misconfigurations, unsafe API usage, and insecure imports across various model repositories. Findings were grouped by CWE and severity level to facilitate cross-tool comparison.

4.5 Platform Mitigation

To address this research question, we conducted a systematic inspection of each model hub to identify the mechanisms implemented to mitigate unsafe model loading. First, we analyzed the official documentation, developer guides, and API references for all platforms to understand their stated security policies and recommendations for custom code execution. Second, we reviewed their open-source repositories (where available) to verify the presence and enforcement of safety-related configurations, including parameters such as `trust_remote_code` or `trust_repo`. Third, we examined whether the platforms perform static or dynamic safety checks before model loading, display explicit warning prompts to users, or isolate model execution within sandboxed or containerized environments. Finally, we compared these mechanisms across platforms to assess their consistency, enforcement levels, and potential gaps in protecting developers and end users from arbitrary code-execution risks.

4.6 Developers' Concerns

To answer this research question, we focused on the Hugging Face and PyTorch Hub forums, GitHub discussions, pull requests, issues, and the Stack Overflow Q&A platform. We searched these platforms using the keywords `trust_remote_code` and `trust_repo`. We identified a total of 418 Hugging Face forum posts, 354 GitHub discussion posts, and 289 Stack Overflow posts. For GitHub issues, there are more than 13,000, and for GitHub pull requests, more than 4,000, using the given query. Hence, we only keep issues and pull requests that include any of the keywords in their title. After filtering for relevance to our study, we retained 27 GitHub discussions, 222 GitHub issues, 297 GitHub pull requests, 27 Stack Overflow posts, and 39 Hugging Face discussions. However, we collected 23 discussions from the PyTorch Hub forums, but none were relevant to our study.

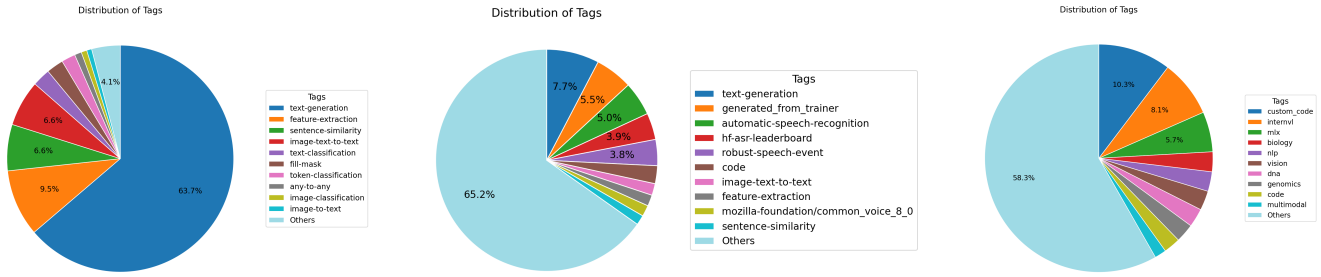

Figure 2: Tag distributions of the custom models. Need to fix the figure aspect ratio.

We then applied an *open coding* approach to the selected posts, carefully reading, analyzing, and annotating each post with conceptual labels (*codes*) reflecting developers' expressed sentiments, challenges, and misunderstandings. The coding was conducted collaboratively by the authors, whose software development experience ranged from 4 to 12 years. We iteratively reviewed and refined the codes through regular calibration meetings until conceptual saturation was achieved.

5 Results

5.1 RQ1: Prevalence of Custom Models

In Table 2, we provide a summary of the collected repositories containing custom code. For Hugging Face, OpenCSG, and ModelScope, approximately 2% to 4% of the models include custom code. In contrast, for OpenMMLab and PyTorch Hub, all available repositories rely on custom code. One notable distinction between these two groups is the verification model: in Hugging Face, OpenCSG, and ModelScope, any user can freely upload models with custom code, whereas in OpenMMLab and PyTorch Hub, models are verified and curated manually by platform maintainers.

Table 2: Result of collected repositories with custom codes.

Platform	# Models Repos	# Collected Repos	(%)	Supported Libraries
Hugging Face	1,699,968	35,953	2.11%	`transformers`
OpenCSG	192,556	6165	3.20%	`transformers`
ModelScope	68,736	3193	4.65%	`PyTorch & transformers`
OpenMMLab	16	16	100.00%	`MMengine & PyTorch`
PyTorch Hub	26	26	100.00%	`PyTorch`

In Figure 2, we present the top 10 tag distributions of custom models across Hugging Face, ModelScope, and OpenCSG platforms. For Hugging Face, the majority of custom models are used for *text generation* (63.7%), followed by *feature extraction* (approximately 10%). In contrast, for OpenCSG, around 7.7% of the custom models are used for text generation, while roughly 8% are associated with *speech-related* tasks. Finally, for ModelScope, the distribution indicates that most custom models are concentrated in *scientific and domain-specific applications*, reflecting its distinct usage patterns compared to the other two platforms. For PyTorch Hub, most of the models focus on vision and audio-related tasks, and for OpenMM Lab, model repositories mainly contain custom models for vision-related tasks.

5.2 RQ2: Presence of security smells and vulnerabilities

Table 3 presents the distribution of Common Weakness Enumerations (CWEs) and the top Bandit issues by severity across five major model-sharing platforms. The results reveal that security weaknesses are *widespread but unevenly distributed* among platforms, with certain categories dominating depending on the ecosystem.

Overall Trends. Across all platforms, **CWE-703 (Improper Check or Handling of Exceptional Conditions)** is by far the most prevalent weakness, particularly on Hugging Face (78.00%) and ModelScope (80.95%), indicating that model repositories frequently include weak or missing exception handling logic. This pattern is strongly associated with the Bandit low-severity rule **B101 (assert statements)**, which alone accounts for 60–80% of all detected issues on most platforms. While these may not directly expose models to immediate exploits, they represent fragile or insecure coding practices that can lead to reliability and maintainability problems.

In the following, we provide platform-specific observation:

Hugging Face. Out of 35,953 repositories with custom codes, 3,743 (10.41%) exhibited at least one security smell. In addition to CWE-703, CWE-494 (Download of Code Without Integrity Check) and CWE-259 (Use of Hard-coded Password) were common. Medium-severity issues were dominated by unsafe Hugging Face Hub downloads (B615, 8.01%) and unsafe PyTorch loading (B614, 2.39%). Although high-severity issues such as starting shell processes (B605) were rare, their presence in community-contributed models is a significant concern.

OpenCSG. Despite a relatively modest smelly repo rate of 2.34%, the absolute number of issues is the highest among all platforms, reflecting the massive scale of the ecosystem (192,556 repositories). Unsafe downloads (B615, 15.11%) and unsafe PyTorch loads (B614, 5.81%) were particularly widespread, suggesting weak trust boundaries and a lack of rigorous review before publication.

ModelScope. This platform shows the lowest proportion of affected repositories (0.33%), yet a similar CWE profile dominated by CWE-703 and CWE-259. The high prevalence of B101 issues indicates poor exception-handling practices, even in repositories that otherwise follow more controlled upload processes.

Table 3: CWE distribution and top Bandit issues per severity across platforms.

Platform	# Repo Analyzed	# Smelly Repo (%)	Top 5 CWEs	Top 2 Issues Per Severity
Hugging Face	35,953	3,743 (10.41%)	CWE-703: 160,967 (78.09%) CWE-494: 19,840 (9.62%) CWE-259: 13,919 (6.75%) CWE-502: 6,136 (2.98%) CWE-78: 5,268 (2.56%)	**HIGH**: 1. B605 — Starting a process with a shell. (394; 0.19%) 2. B602 — subprocess call with `shell=True`. (87; 0.04%) **MEDIUM**: 1. B615 — Unsafe Hugging Face Hub download. (17,042; 8.01%) 2. B614 — Use of unsafe PyTorch load. (5,086; 2.39%) **LOW**: 1. B101 — Use of assert detected. (157,788; 74.18%) 2. B107 — Possible hardcoded password. (3,825; 1.80%)
OpenCSG	192,556	4,503 (2.34%)	CWE-703: 40,433 (64.55%) CWE-494: 13,298 (21.23%) CWE-502: 4,426 (7.07%) CWE-78: 2,250 (3.59%) CWE-259: 2,230 (3.56%)	**HIGH**: 1. B605 — Starting a process with a shell. (221; 0.34%) 2. B602 — `subprocess` with `shell=true`. (179; 0.27%) **MEDIUM**: 1. B615 — Unsafe Hugging Face Hub download. (9,957; 15.11%) 2. B614 — Use of unsafe PyTorch load. (3,825; 5.81%) **LOW**: 1. B101 — Use of assert detected. (39,717; 60.28%) 2. B311 — Pseudorandom generator used for security. (2,154; 3.27%)
ModelScope	68,736	229 (0.33%)	CWE-703: 20,490 (80.95%) CWE-259: 2,584 (10.21%) CWE-494: 1,455 (5.75%) CWE-502: 464 (1.83%) CWE-78: 320 (1.26%)	**HIGH**: 1. B605 — Starting a process with a shell. (22; 0.09%) 2. B324 — Use of weak MD5 hash for security. (18; 0.07%) **MEDIUM**: 1. B615 — Unsafe Hugging Face Hub download. (1,278; 4.95%) 2. B614 — Use of unsafe PyTorch load. (421; 1.63%) **LOW**: 1. B101 — Use of assert detected. (20,169; 78.17%) 2. B107 — Possible hardcoded password. (875; 3.39%)
OpenMM	16	12 (75.00%)	CWE-703: 848 (86.44%) CWE-259: 124 (12.64%) CWE-502: 6 (0.61%) CWE-78: 3 (0.31%)	**HIGH**: — **MEDIUM**: 1. B301 — `pickle` used to deserialize untrusted data. (4; 0.41%) 2. B307 — Use of `eval`. (2; 0.20%) **LOW**: 1. B101 — Use of assert detected. (848; 86.44%) 2. B106 — Possible hardcoded password. (59; 6.01%)
PyTorch Hub	26	12 (46.15%)	CWE-78: 42 (35.90%) CWE-703: 40 (34.19%) CWE-502: 22 (18.80%) CWE-330: 12 (10.26%) CWE-22: 1 (0.85%)	**HIGH**: 1. B602 — `subprocess` with `shell=true`. (4; 3.39%) **MEDIUM**: 1. B614 — Use of unsafe PyTorch load. (19; 16.10%) 2. B307 — Use of `eval`. (16; 13.56%) **LOW**: 1. B101 — Use of assert detected. (38; 32.20%) 2. B311 — Pseudorandom generator used for security. (12; 10.17%)

OpenMMLab. Although the total number of repositories is small (16), 75% contained security smells. Here, CWE-703 accounted for 86.44% of issues, and unsafe deserialization (B301) and `eval` usage (B307) were notable medium-severity findings, both of which are associated with high-impact attack vectors if exploited.

PyTorch Hub. Nearly half of the repositories (46.15%) contained security issues. Unlike other platforms, CWE-78 (OS Command Injection) was the most common weakness, reflecting the frequent use of shell commands and `eval` functions. The presence of B602 (subprocess with `shell=true`) and B307 (eval usage) further confirms that this platform is particularly susceptible to remote code execution risks.

Table 4 summarizes the distribution of Common Weakness Enumerations (CWEs), top OWASP categories, and the most frequent Semgrep rule violations across major model-sharing platforms. In contrast to Bandit results, which primarily surfaced lower-severity coding smells, the Semgrep analysis reveals a *clear concentration of security-critical issues*, particularly related to unsafe deserialization, code injection, and integrity failures.

Overall Trends. Across all platforms, **CWE-502 (Deserialization of Untrusted Data)** is the most frequent weakness, consistently appearing in 50–80% of the flagged repositories. This highlights the pervasiveness of unsafe serialization/deserialization practices in model loading pipelines, where frameworks like PyTorch often rely on `pickle`-based mechanisms. Additionally, CWE-95 (Eval Injection), CWE-676 (Use of Potentially Dangerous Function), and CWE-706 (Improper Handling of Variadic Functions) appear across platforms, indicating multiple attack surfaces related to dynamic code execution and weak input handling.

The top OWASP categories identified through Semgrep closely align with injection-based threats. **Injection** vulnerabilities dominate on most platforms (e.g., 54.6% on Hugging Face and 67.9% on PyTorch Hub), followed by **Insecure Deserialization** and **Integrity Failures**. These categories indicate that many repositories contain code that can be abused to execute arbitrary payloads at load time or bypass expected trust boundaries. Broken Access Control and Cryptographic Failures were less frequent but still notable, particularly on OpenCSG.

Table 4: CWE distribution and top Semgrep issues per severity across platforms.

Platform	# Repo Analyzed	# Smelly Repo (%)	Top 5 CWEs	Top 5 OWASP	Top 5 Rules
Hugging Face	36,697	461 (1.26%)	CWE-502: 462 (64.26%) CWE-95: 183 (25.45%) CWE-676: 29 (4.03%) CWE-942: 26 (3.62%) CWE-706: 19 (2.64%)	Injection: 190 (54.60%) Insecure Deserialization: 56 (16.09%) Integrity Failures: 56 (16.09%) Security Misconfiguration: 26 (7.47%) Broken Access Control: 20 (5.75%)	pickles in pytorch: 406 (41.90%) numpy in pytorch: 296 (30.55%) eval detected: 182 (18.78%) avoid pickle: 56 (5.78%) automatic memory pinning: 29 (2.99%)
OpenCSG	192,556	1,141 (0.59%)	CWE-502: 5,449 (82.81%) CWE-676: 451 (6.85%) CWE-95: 344 (5.23%) CWE-706: 180 (2.74%) CWE-78: 156 (2.37%)	Injection: 616 (30.36%) Integrity Failures: 493 (24.30%) Insecure Deserialization: 416 (20.50%) Broken Access Control: 261 (12.86%) Cryptographic Failures: 243 (11.98%)	pickles in pytorch: 4,923 (67.73%) numpy in pytorch: 1,279 (17.60%) automatic memory pinning: 414 (5.70%) avoid pickle: 379 (5.21%) eval detected: 274 (3.77%)
ModelScope	68,736	581 (0.85%)	CWE-502: 526 (72.65%) CWE-95: 144 (19.89%) CWE-706: 28 (3.87%) CWE-676: 14 (1.93%) CWE-96: 12 (1.66%)	Injection: 159 (58.46%) Insecure Deserialization: 38 (13.97%) Integrity Failures: 38 (13.97%) Broken Access Control: 29 (10.66%) Sensitive Data Exposure: 8 (2.94%)	pickles in pytorch: 484 (46.45%) numpy in pytorch: 351 (33.69%) eval detected: 143 (13.72%) avoid pickle: 36 (3.45%) non literal import: 28 (2.69%)
OpenMM	16	2 (12.50%)	CWE-502: 8 (61.54%) CWE-95: 3 (23.08%) CWE-706: 2 (15.38%)	Insecure Deserialization: 8 (38.10%) Integrity Failures: 8 (38.10%) Injection: 3 (14.29%) Broken Access Control: 2 (9.52%)	avoid pickle: 8 (61.54%) eval detected: 2 (15.38%) non-literal import: 2 (15.38%) exec detected: 1 (7.69%)
PyTorch Hub	26	10 (38.46%)	CWE-502: 25 (49.02%) CWE-95: 16 (31.37%) CWE-676: 5 (9.80%) CWE-78: 3 (5.88%) CWE-706: 2 (3.92%)	Injection (A03:21): 19 (67.86%) Injection (A01:17): 3 (10.71%) Insecure Deserialization: 2 (7.14%) Integrity Failures: 2 (7.14%) Broken Access Control: 2 (7.14%)	pickles in pytorch: 23 (46.94%) eval detected: 16 (32.65%) automatic memory pinning: 5 (10.20%) subprocess shell true: 3 (6.12%) avoid pickle: 2 (4.08%)

In the following, we provide platform-specific observation:

Hugging Face. Affected 1.26% of repositories, with CWE-502 and CWE-95 dominating. Injection-related patterns and `pickle` usage are particularly common, indicating elevated risk from untrusted model artifacts.

OpenCSG. Despite a lower proportion of smelly repos (0.59%), the absolute number of issues is large due to the ecosystem scale. Unsafe pickle usage (67.7%) and integrity failures stand out as systemic weaknesses.

ModelScope. CWE-502 (72.65%) remains the leading weakness, with injection issues concentrated around eval usage and unsafe deserialization patterns.

OpenMMLab. Though small in absolute numbers, the issues are concentrated and severe, with avoid-pickle violations and eval usage indicating poor deserialization hygiene.

PyTorch Hub. A striking 38.46% of repositories are affected, dominated by injection (67.86%) and eval-based code execution patterns (32.65%). This confirms that PyTorch Hub exposes a high-risk attack surface.

5.3 RQ3: Platform Mitigation Strategies

We evaluated the security mechanisms across five platforms. Table 5 summarizes the key features of each platform. All claims were verified against official documentation as of October 2025. "None documented" indicates that no information was found in publicly available official documentation and does not imply the definitive absence of the feature.

5.3.1 Warning Systems and User Notifications. Hugging Face implements the most comprehensive warning system among the platforms studied, displaying banners when the models require `trust_remote_code=True`. These warnings appear in both the web interface and the Transformers library programmatically. In addition, the model card on Hugging Face allows users to directly copy code snippets for using the model with `transformers`, but it provides no warning about enabling `trust_remote_code=True` or suggestion about pinning to a specific revision.

ModelScope provides similar warnings, but with less prominence in the UI. Although OpenCSG supports `transformer` libraries, it does not explicitly mention the similar warning banner when `trust_remote_code=True` is used in their documentation. OpenMM, in this case, is keeping its model zoo closed, while supporting custom models and welcoming pull requests. These are reviewed, and therefore, no warning is shown when using their models. In contrast to the `trust_remote_code` flag, PyTorch Hub's `trust_repo` flag operates more discreetly, with minimal user-facing warnings beyond the documentation. Moreover, if it is set to false, a prompt will ask the user whether the repository should be trusted.

5.3.2 Technical Safeguards and Verification. Critical security gaps exist across all platforms. No platform implements comprehensive sandboxing for custom code execution during model loading. Although Hugging Face has announced plans for isolated execution environments, as of our study period, custom code runs with the same privileges as the loading application. Only Hugging Face operates automated scanning pipelines across uploaded repositories, combining internal malware and pickle detectors, secrets scanning, and third-party tools such as Protect AI's Guardian. These

Table 5: Comparison of security and trust mechanisms across model-sharing platforms. All claims verified against official documentation.

Platform	Upload Verification	Trust Model	Malware Scanning	Warning Systems	User Protection
Hugging Face	Automated pipeline on every push: ClamAV (malware), PickleScan (pickle/RCE), TruffleHog (secrets) [8, 9]; plus third-party scanners (Protect AI Guardian, JFrog) [13]	Trust-all with verified badges [11]	Yes – comprehensive multi-layered [2, 10]	UI warnings, file badges (ok/infected), & verified badges [10]	Documentation hub [12], UI warnings, SafeTensors support [8]
OpenCSG	Open uploads via Git or web [31]; optional content moderation (text/image) [28]	Community trust [29]	None documented [30]	None documented [30]	Documentation (community guidelines) [30]
ModelScope	No platform-level automated scanning documented [25]	Trust-all (no verification) [26]	None – no platform-level scanning [25]	Post-download `trust_remote_code` warnings only [3, 27]	Documentation, `trust_remote_code` parameter [27]
OpenMMLab	Maintainer pull-request review [37]	Verify-first [37]	None documented [36]	None documented [36]	Documentation (reviewed code) [38]
PyTorch Hub	No automated verification; `skip_validation` for repo ownership checks [42]	Trust-all with `trust_repo` parameter [42]	None documented [42]	Interactive prompts (`trust_repo`), deprecation warnings [42, 43]	Strong documentation warnings: "models are programs" [41]

pipelines trigger on every push/commit and when repository pages are loaded, but they target known patterns and file types rather than offering full static analysis of arbitrary custom code, so residual RCE risk remains. [10, 14, 19, 40]

5.3.3 Alternative Secure Loading Mechanisms. Platforms are increasingly offering alternatives to custom code execution, although adoption remains limited. Several platforms now support static model formats that eliminate the need for custom code, including Safetensors (supported by Hugging Face), ONNX (cross-platform format), and TensorFlow Lite. Despite the availability of secure alternatives, adoption remains critically low: an August 2025 ecosystem snapshot of 1.86 million Hugging Face models found that only 6.6% of models with weights used the Safetensors format. [24]

5.3.4 Evolution of Platform Security Infrastructure (2024-2025). Recent developments in platform security infrastructure reveal significant disparities in industry maturity. Hugging Face has established the industry benchmark through a multi-layered security architecture that, by April 2025, had scanned 4.47 million unique model versions across 1.41 million repositories and was processing 226 million security scan requests monthly [40]. The platform's October 2024 partnership with Protect AI's Guardian platform delivered new ML-specific detection modules (e.g., PAIT-ARV-100 for archive slip, PAIT-TF-200 for TensorFlow backdoors, and PAIT-LMAFL-300 for Llamafile runtime risks) that surface inline alerts and detailed vulnerability reports to repository maintainers [40]. The August 2025 Cisco partnership unified malware scanning through ClamAV 1.5 with AI-specific threat detection capabilities [2].

In stark contrast, ModelScope exposes no documented platform-level malware scanning or automated verification for repository uploads, retaining only post-download warnings [25]. OpenCSG likewise allows unrestricted code/model uploads without advertising Hub-wide malware scanning, although its security FAQs describe automated MCP scanning that fires during agent updates or calls and developer-centric SecScan/CodeSouler tooling for detecting poisoned toolchains and vulnerable dependencies—protections scoped to agent workflows rather than default repository screening [28, 31–35]. PyTorch Hub similarly provides minimal automated protections, with the `trust_repo` parameter offering only interactive prompts without comprehensive scanning infrastructure [42, 43].

The emergence of SafeTensors as a secure serialization standard represents a critical security advancement. By storing tensors in a simple, data-only layout (rather than pickle-based structures) Safetensors prevents executable payloads from hitching a ride with model weights while still enabling zero-copy loading for performance-sensitive serving. [18] Hugging Face's Text Generation Inference stack therefore expects Safetensors artifacts by default and converts PyTorch checkpoints automatically when none are present, aligning the serving ecosystem around the safer format. [17] Yet ecosystem-wide measurements show that only 6.6% of models with weights ship in this format, leaving substantial room for improvement despite platform nudges and automated conversions. [17, 24]

5.4 Developers' Concern

In Figure 3, we provided the taxonomy developed from different individual platforms.

6 Discussion

6.1 Ecosystem-wide Security Exposure

Our findings reveal that the **model-sharing ecosystem is broadly and unevenly exposed to security risks**. As shown in Table 2, while only 2-4% of models on platforms such as Hugging Face, ModelScope, and OpenCSG require custom code; this seemingly small subset represents around *45,000 repositories* containing code executed at load time. Platforms such as OpenMMLab and PyTorch Hub rely entirely on custom code, increasing their systemic attack surface.

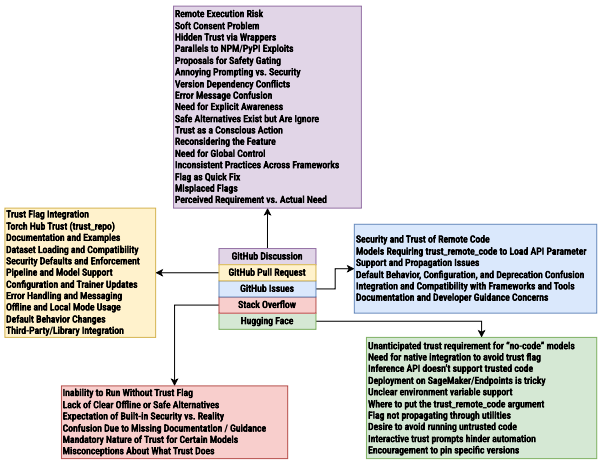

Figure 3: Taxonomy of Developers' Concerns.

Static analysis with Bandit and Semgrep identified two major vulnerability clusters. First, **low-severity but pervasive coding smells** (e.g., CWE-703, B101 `assert` statements) appear across 60–80% of affected repositories, reflecting weak defensive programming practices. Second, **high-impact injection and deserialization vulnerabilities** (e.g., CWE-502, CWE-95, CWE-78) were widespread, particularly on PyTorch Hub and Hugging Face, where dynamic code execution via `pickle` and `eval` is common. Semgrep analysis identified injection and insecure deserialization as the top OWASP categories, underscoring systemic risks of arbitrary code execution at model load time.

Importantly, these exposures are not uniform. OpenCSG, despite a low percentage of smelly repos, contributes the highest absolute number of issues due to its massive scale. PyTorch Hub, with a smaller ecosystem, has a disproportionately high rate of high-severity issues, highlighting differences in platform trust boundaries and code review practices.

6.2 Gaps Between Security Mechanisms and Developer Practices

The analysis of platform security mechanisms (Table 5) highlights a clear **misalignment between available safeguards and how developers interact with model repositories**. Hugging Face, for instance, operates the most advanced malware scanning pipeline — integrating ClamAV, PickleScan, TruffleHog, and Protect AI Guardian—yet *unsafe practices persist widely*, including reliance on `pickle` serialization and unpinned revision loading. The presence of CWE-502 and CWE-95 in hundreds of repositories demonstrates that **technical defenses alone are insufficient to change developer behavior**.

Similarly, ModelScope issues warning banners for `trust_remote_code` but lacks sandboxing or pre-upload verification, allowing risky code to propagate. OpenCSG and PyTorch Hub provide minimal automated scanning, relying instead on community trust or basic user prompts (`trust_repo`). The high concentration of injection- and eval-based vulnerabilities in the PyTorch Hub underscores the risks of such lightweight defenses. Furthermore, the *low adoption of secure formats such as Safetensors* (only 6.6% on Hugging Face as of August 2025) shows that safer alternatives are not being embraced at scale, often due to developer inertia, ecosystem lock-in, or lack of clear incentives.

6.3 Implications for Research and Practitioners

These findings have critical implications for both security research and practitioner communities.

For Platform Operators. Platforms must move beyond passive warning systems toward **enforced security boundaries**, including default sandboxing of untrusted custom code, mandatory integrity checks, and stricter upload verification workflows. Richer developer-facing telemetry (e.g., inline vulnerability alerts, dependency provenance) can bridge the gap between automated scanning and practical adoption of secure practices.

For Developers and Maintainers. The results emphasize that developers play a decisive role in the security posture of model hubs. Reliance on `pickle` and eval-based code should be minimized or replaced with safe loading alternatives. Incorporating secure defaults, revision pinning, and code review checklists can help reduce CWE hotspots such as CWE-502 and CWE-95.

For Researchers. Our work showed that though platforms used shared libraries underneath, they have significant differences in handling custom code in model loading. There needs to be work on **automated enforcement frameworks** for trust boundaries, integrating cryptographic integrity verification with runtime isolation. We also need to study safe alternatives and their adoption of security warnings and secure model formats (e.g., Safetensors, ONNX).

Ultimately, **model repositories must be treated as software supply chains**. Just as package managers evolved to include signing, scanning, and enforcement layers, model hubs must integrate similar mechanisms by default. Our results highlight that while some platforms, notably Hugging Face, have taken meaningful steps, the ecosystem at large remains *undersecured and highly exposed*, requiring coordinated action from platform maintainers, security researchers, and practitioners alike.

6.4 Threats of Validity

Our study involves large-scale measurement and security analysis of model-sharing platforms. While we applied systematic methods to minimize bias and maximize coverage, several threats to validity may affect the interpretation and generalizability of our findings.

Internal Validity. A primary internal threat lies in the accuracy and completeness of our static analysis. Although we employed three well-established tools—Bandit, CodeQL, and Semgrep—to identify security smells and CWE patterns, they may produce false positives or false negatives. However, Siddiq *et al.* show Bandit has 90.79% precision. Semgrep and CodeQL are widely used in the research community. citation will be added Another thing is that

we manually analyze discussion posts to conduct open-coding. As mentioned before, this coding was conducted collaboratively by the authors, whose software development experience ranged from 4 to 12 years. An experienced author mitigated the discrepancy. cohen-kappa will be added later

External Validity. Our results may not fully generalize beyond the platforms studied. We focused on five major platforms—Hugging Face, OpenCSG, ModelScope, OpenMMLab, and PyTorch Hub—that dominate the model-sharing ecosystem. For example, Hugging Face hosts around 1.7 million models, and OpenCSG hosts around 200k models. Smaller or private repositories (e.g., enterprise model registries) may exhibit different security characteristics.

7 Related Work

7.1 Evolution of Model Hosting Platforms and Pipelines

The evolution from localized model development to centralized sharing platforms marks a fundamental shift in collaborative machine learning (ML) practices. In the early stages, researchers relied on manual distribution via institutional websites or GitHub repositories, which required end users to reconstruct the entire training and execution environment to reproduce results. The first generation of organized model distribution are mainly *Caffe* (2014) and *TensorFlow Hub* (2018). These platforms functioned primarily as static catalogs of pretrained models.

A more profound transformation began with *PyTorch Hub* in 2019, which introduced the `torch.hub.load()` interface along with the `trust_repo` parameter [42]. The rapid expansion of *Hugging Face* between 2018 and 2023 further reshaped the landscape: evolving from a simple hosting repository to a fully integrated platform supporting training, inference, and deployment workflows. Yi *et al.* [48] provides a comprehensive analysis of this ecosystem's development, showing how model hubs have become critical infrastructure sustaining millions of models and billions of downloads worldwide. Their supply chain–oriented perspective reveals how LLM-integrated platforms introduce cascading security risks that traditional software assurance models fail to capture. This infrastructural transformation is further quantified by Laufer *et al.* , by analyzing two million models hosted on Hugging Face [24]. Their findings highlight the platform's support for over 4,000 distinct architectures, with an increasing proportion of models depending on custom code to enable advanced functionality beyond standard implementations. Our work focuses on the architectural design of executing code during model loading from the hub.

7.2 Quality and security issues of Model Hubs

Jiang et al. [21] conducted the first systematic study of these artifacts across eight platforms, revealing that trust relationships in model ecosystems are more implicit and poorly understood than in traditional software. They found that users often conflate platform reputation with individual model safety, a misconception that platforms inadvertently reinforce through download counts and trending metrics. Recent surveys extend this analysis to large language model pipelines specifically. Hu et al. [7] identifies open problems in the LLM supply chain, documenting how fine-tuning workflows, adapter layers, and prompt templates all serve as injection points. Yi et al. [48] further characterizes these risks from an edge computing perspective, showing how LLM-integrated platforms create new trust boundaries between cloud services, edge devices, and end users.

The pickle protocol, ubiquitous in Python ML frameworks, fundamentally operates as a stack-based virtual machine that executes bytecode during deserialization. When PyTorch calls `torch.load()`, it invokes pickle's `Unpickler`, which processes opcodes that can instantiate arbitrary classes and call methods through the `__reduce__` protocol [41]. This design choice, while enabling complex object serialization, violates the principle of least privilege. The PyTorch security documentation explicitly warns that "pickle is not secure" and that `torch.load()` should only be used with trusted data [41]. The introduction of `weights_only=True` in PyTorch 1.13 [43] and the `trust_remote_code` flag in Transformers 4.0 [15] represent acknowledgments of these risks, but adoption remains low due to compatibility concerns. Alternative serialization approaches demonstrate different security-functionality tradeoffs. SafeTensors, introduced by Hugging Face in 2022 [18], uses a simple header-data format that prevents code execution entirely. The format stores tensors in a flat layout with minimal metadata, enabling zero-copy loading while eliminating executable payloads. However, as our results show and Laufer et al. confirm [24], only 6.6% of models have adopted this format despite platform encouragement. Recent work on secure deserialization provides partial solutions. PickleBall [23] introduces semantics-aware loading that permits safe pickle subsets through allowlisting specific opcodes and validating operation sequences. Related vulnerabilities in other frameworks—such as CERT/CC's advisory on Keras Lambda layers (CVE-2024-3660) [1], demonstrate that the problem extends beyond pickle to any format that conflates data with executable logic.

While platform owners use scanners to find out vulnerable codes and data, Zhao et al.'s deployment of *MalHug* [49] found out 91 malicious models and 9 dataset scripts actively exploiting users. JFrog Security Research [20] documented evasion methods that bypass pattern-based scanning, including time-delayed execution, environment fingerprinting, and polymorphic code generation. The August 2025 Protect AI report [40], based on scanning 4.47 million model versions, identifies emerging threats, such as archive slip vulnerabilities and TensorFlow-specific backdoors, that existing tools miss. Our work specifically focuses on the code associated with the model, which is executed during loading, and on developers' concerns about it.

8 Conclusion

Our work provides the first large-scale, cross-platform empirical analysis of remote code execution risks in ML model hosting ecosystems, examining five major platforms—Hugging Face, OpenCSG, ModelScope, OpenMMLab, and PyTorch Hub. We identified that while only a small fraction (2–4%) of models on some platforms

require custom code, this still translates to around 45,000 repositories that execute arbitrary code during model loading. On platforms like OpenMMLab and PyTorch Hub, this figure reaches 100%, revealing a deeply embedded systemic risk. Our static analysis with Bandit, CodeQL, and Semgrep revealed two dominant vulnerability clusters: (1) Pervasive low-severity coding smells (e.g., CWE-703) across most repositories, indicating weak defensive coding practices, and (2) High-severity injection and deserialization vulnerabilities (e.g., CWE-502, CWE-95, CWE-78), concentrated particularly on PyTorch Hub and Hugging Face. Although Hugging Face has made significant advances with automated malware scanning pipelines (e.g., ClamAV, PickleScan, Protect AI Guardian), these mechanisms alone are insufficient. Other platforms lack comparable safeguards, with minimal or no sandboxing and weak verification mechanisms. Developer discussions further reveal widespread confusion and misconceptions about trust flags, limited adoption of secure serialization formats like SafeTensors, and tension between usability and security. In addition, without enough documentation, developers feel confused while adopting the platform's security mechanisms. In the future, we will focus on an automated enforcement framework for trust boundaries, integrating cryptographic integrity verification with runtime isolation for remote code execution during model loading. We will also explore safe alternatives to custom model loading.

References

[1] CERT Coordination Center. 2024. VU#253266 – Keras Lambda Layers Allow Arbitrary Code Execution. https://kb.cert.org/vuls/id/253266. Accessed: 2025-10-10.

[2] Cisco. 2024. Foundation AI Advances AI Security With Hugging Face. https://blogs.cisco.com/security/ciscos-foundation-ai-advances-ai-supply-chain-security-with-hugging-face. Accessed: 2025-10-10.

[3] EvalScope. 2024. FAQ. https://evalscope.readthedocs.io/en/v0.16.3/get_started/faq.html. Accessed: 2025-10-10.

[4] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA.

[5] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. 2017. Security smells in android. In *2017 IEEE 17th international working conference on source code analysis and manipulation (SCAM)*. IEEE, 121–130. doi:10.1109/SCAM.2017.24

[6] Aakanshi Gupta, Bharti Suri, Vijay Kumar, Sanjay Misra, Tomas Blažauskas, and Robertas Damaševičius. 2018. Software code smell prediction model using Shannon, Rényi and Tsallis entropies. *Entropy* 20, 5 (2018), 372.

[7] Zuxin Hu, Ning Liu, Tian Zhao, Fan Yang, Xiaowei Deng, Ting Du, Jian Chen, Zongwei Li, and Xiaolong Fan. 2025. Large Language Model Supply Chain: Open Problems from the Security Perspective. In *Proceedings of the 32nd ACM Conference on Computer and Communications Security*. Association for Computing Machinery. doi:10.1145/3713081.3731747

[8] Hugging Face. 2024. 2024 Security Feature Highlights. https://huggingface.co/blog/2024-security-features. Accessed: 2025-10-10.

[9] Hugging Face. 2024. Hugging Face Partners with TruffleHog to Scan for Secrets. https://huggingface.co/blog/trufflesecurity-partnership. Accessed: 2025-10-10.

[10] Hugging Face. 2024. Malware Scanning. https://huggingface.co/docs/hub/en/security-malware. Accessed: 2025-10-10.

[11] Hugging Face. 2024. Organization Verification. https://discuss.huggingface.co/t/organization-verification/17906. Accessed: 2025-10-10.

[12] Hugging Face. 2024. Security. https://huggingface.co/docs/hub/en/security. Accessed: 2025-10-10.

[13] Hugging Face. 2024. Security & Compliance. https://huggingface.co/docs/microsoft-azure/en/security. Accessed: 2025-10-10.

[14] Hugging Face. 2024. Third-party scanner: Protect AI. https://huggingface.co/docs/hub/en/security-protectai. Accessed: 2025-10-10.

[15] Hugging Face. 2025. Auto Classes. https://huggingface.co/docs/transformers/en/model_doc/auto. Accessed: 2025-10-10.

[16] Hugging Face. 2025. Customizing models — Transformers Documentation. https://huggingface.co/docs/transformers/en/custom_models Accessed: 2025-10-13.

[17] Hugging Face. 2025. Safetensors. https://huggingface.co/docs/text-generation-inference/en/conceptual/safetensors. Accessed: 2025-10-10.

[18] Hugging Face. 2025. Safetensors Documentation. https://huggingface.co/docs/safetensors/index. Accessed: 2025-10-10.

[19] Hugging Face. 2025. Secrets Scanning. https://huggingface.co/docs/hub/en/security-secrets. Accessed: 2025-10-10.

[20] JFrog Security Research. 2024. Examining Malicious Hugging Face ML Models with Silent Backdoors. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/. Accessed: 2025-10-10.

[21] Shangqing Jiang, Nicholas Synovic, Chahat Sethi, Sandeep Indarapu, Parker Hyatt, Marco Schorlemmer, George Thiruvathukal, and James C. Davis. 2022. An Empirical Study of Artifacts and Security Risks in the Pre-trained Model Supply Chain. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. Association for Computing Machinery, 105–114. doi:10.1145/3560835.3564547

[22] Stephen H Kan. 2003. *Metrics and models in software quality engineering*. Addison-Wesley Professional.

[23] Andreas D. Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P. Kemerlis, James C. Davis, and Junfeng Yang. 2025. PickleBall: Secure Deserialization of Pickle-based Machine Learning Models (Extended Report). https://arxiv.org/abs/2508.15987. arXiv:2508.15987.

[24] Benjamin Laufer, Hamidah Oderinwale, and Jon Kleinberg. 2025. Anatomy of a Machine Learning Ecosystem: 2 Million Models on Hugging Face. *arXiv* (2025). arXiv:2508.06811 [cs.SI] https://arxiv.org/abs/2508.06811 Accessed: 2025-10-10.

[25] ModelScope. 2024. ModelScope: Bring the Notion of Model-as-a-Service to Life. https://github.com/modelscope/modelscope. Accessed: 2025-10-10.

[26] ModelScope. 2024. Phi-3-mini-128k-instruct. https://modelscope.cn/models/LLM-Research/Phi-3-mini-128k-instruct. Accessed: 2025-10-10.

[27] ModelScope. 2024. Releases. https://github.com/modelscope/modelscope/releases. Accessed: 2025-10-10.

[28] OpenCSG. 2024. csghub-server. https://github.com/OpenCSGs/csghub-server. Accessed: 2025-10-10.

[29] OpenCSG. 2024. Information about the OpenCSG community. https://github.com/OpenCSGs/community. Accessed: 2025-10-10.

[30] OpenCSG. 2024. OpenCSG Documentation Center. https://www.opencsg.com/docs/en/. Accessed: 2025-10-10.

[31] OpenCSG. 2024. Uploading Codes. https://www.opencsg.com/docs/en/code/upload_codes. Accessed: 2025-10-10.

[32] OpenCSG. 2025. CSGShip CodeSouler—Your Intelligent Coding Companion. https://www.opencsg.com/docs/en/starship/codesouler/codesouler_intro. Accessed: 2025-10-10.

[33] OpenCSG. 2025. FAQs. https://www.opencsg.com/docs/en/q%26a. Accessed: 2025-10-10.

[34] OpenCSG. 2025. The ROI of Control: Calculating the True Cost of Your Hugging Face Strategy. https://medium.com/@OpenCSG/the-roi-of-control-calculating-the-true-cost-of-your-hugging-face-strategy-de41824c6da3. Accessed: 2025-10-10.

[35] OpenCSG. 2025. StarShip SecScan. https://www.opencsg.com/docs/en/starship/secscan/. Accessed: 2025-10-10.

[36] OpenMMLab. 2024. Benchmark and Model Zoo - MMDetection's documentation. https://mmdetection.readthedocs.io/en/latest/model_zoo.html. Accessed: 2025-10-10.

[37] OpenMMLab. 2024. Contribution Guide – MMDetection3D 1.4.0 documentation. https://mmdetection3d.readthedocs.io/en/latest/notes/contribution_guides.html. Accessed: 2025-10-10.

[38] OpenMMLab. 2024. OpenMMLab Detection Toolbox and Benchmark. https://github.com/open-mmlab/mmdetection. Accessed: 2025-10-10.

[39] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering* 29, 1 (Jan. 2022), 47–94. doi:10.1007/s11831-021-09566-x

[40] Protect AI. 2025. 4M Models Scanned: Hugging Face + Protect AI Partnership Update. https://protectai.com/blog/hugging-face-protect-ai-six-months-in. Accessed: 2025-10-10.

[41] PyTorch. 2024. Security Policy - pytorch/pytorch. https://github.com/pytorch/pytorch/security. Accessed: 2025-10-10.

[42] PyTorch. 2024. torch.hub – PyTorch 2.8 documentation. https://pytorch.org/docs/stable/hub.html. Accessed: 2025-10-10.

[43] PyTorch. 2024. torch.load(..., weights_only=True) currently raises a warning. https://github.com/pytorch/pytorch/issues/52181. Accessed: 2025-10-10.

[44] PyTorch. 2025. torch.hub — PyTorch Documentation. https://docs.pytorch.org/docs/stable/hub.html Accessed: 2025-10-13.

[45] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 164–175. doi:10.1109/ICSE.2019.00033

[46] Md Rayhanur Rahman, Akond Rahman, and Laurie Williams. 2019. Share, But be Aware: Security Smells in Python Gists. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 536–540. doi:10.1109/ICSME.

2019.00087

[47] Mohammed Latif Siddiq, Shafayat Hossain Majumder, Maisha Rahman Mim, Sourov Jajodia, and Joanna C.S. Santos. 2022. An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*.

[48] Wenxin Yi, Dinh Phung Luu, Kangyu Ma, Jiayi Wang, Xiaowei Deng, Ruoxi Fu, Jiajing Chen, Zongwei Li, Wenxuan Yu, Yue Wang, Philip Khang, Fan Yang, Yue Zhang, and Yuanyuan Fang. 2024. Characterizing and Understanding the Risks of Large Language Model-Integrated Platforms: A Supply-Chain Perspective on the Security of Edge LLM Systems. *arXiv* (2024). arXiv:2409.09368 [cs.CR] https://arxiv.org/abs/2409.09368 arXiv:2409.09368v1.

[49] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. 2024. Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2087–2098. doi:10.1145/3691620.3695271