SUPPLY-CHAIN ATTACKS IN MACHINE LEARNING FRAMEWORKS

Yue Gao [1] **Ilia Shumailov** [2] **Kassem Fawaz** [1]

ABSTRACT

Machine learning (ML) systems are increasingly vulnerable to supply-chain attacks that exploit the intricate dependencies inherent in open-source software (OSS). However, securing the ML ecosystem remains challenging due to regular paradigmatic changes in the ecosystem, their dynamic runtime environments, and lack of security awareness in open-source ML projects. In this paper, we introduce a novel class of supply-chain attacks that specifically target ML models, relying on inherent insecurity of Python as a programming language. Such attacks leverage traditional supply-chain vulnerabilities to inject innocuous-looking code that weakens the ML model's robustness. We then conduct an LLM-assisted analysis of discussions from the top 50 ML projects on GitHub to understand the current state of supply-chain security awareness among contributors. Despite the need for a higher standard of security practices, our findings reveal a similar level of security awareness between the ML and non-ML communities, highlighting the need for enhanced safeguards against ML-specific supply-chain attacks.

1 INTRODUCTION

Supply-chain attacks have emerged as a critical threat in the development of open-source software (OSS), posing significant risks to the integrity and security of modern software ecosystems (Ladisa et al., 2023). These attacks leverage the intricate dependencies inherent in OSS to inject malicious code, compromising both upstream and downstream applications and users. The machine learning (ML) community, which heavily relies on OSS for rapid innovation and scalability, is particularly vulnerable to such attacks.

This risk is not hypothetical. A PyTorch dependency was compromised with malicious code capable of exfiltrating sensitive data from user machines (PyTorch, 2022), and a vulnerability in the HuggingFace package could lead to unauthorized code execution (Sestito, 2024). ML supply chain compromise can also affect commercial companies and an incident was recently reported at ByteDance caused by an insider (da Silva, 2024). A separate vulnerability was identified in Huggingface's "load by name" feature. This functionality allowed malicious actors to execute "AI-Jacking" attacks by re-registering model names previously associated with legitimate models (Noy, 2023). Recent reports have also highlighted the vulnerability of the PyTorch package to supply chain attacks, exploiting GitHub's CI/CD runners feature to inject malicious code (Young, 2024; Stawinski, 2024).

[1]University of Wisconsin–Madison [2]Google DeepMind. Correspondence to: Yue Gao <gy@cs.wisc.edu>.

Proceedings of the 8^{th} MLSys Conference, Santa Clara, CA, USA, 2025.

Securing the ML ecosystem is challenging due to the highly dynamic nature of the Python runtime environment, which is, at present, foundational to ML development. In a Python code base, any dependency package can access the entire memory space of other packages as well as the calling stack of the downstream applications, thereby reading and modifying any critical variables at runtime. In contrast to languages that prioritize security through explicit features like type safety, compartmentalization, immutability checks, taint analysis, or capability-based security, Python's development ecosystem tends to overlook these safeguards. Because of the lack of such safeguards, a compromised Python package can present significant risks if it is exploited.

Similar to the traditional open-source community, the lack of awareness of supply-chain security among ML project contributors makes securing the ML ecosystem even harder. This is particularly concerning as ML projects usually have a much larger attack surface; importantly, in Section 2.2 we show that ML packages tend to have a lot more dependencies. For example, as a core dependency in many projects, the `requests` package only has 14 direct and transitive dependencies even at the development time. Yet, HuggingFace's `transformers` package currently has **305** of them, adding 20 times more potential entry points for supply-chain attacks. There is also evidence that significant insecure programming practices are normalized for ML supply chains. For example, Huggingface has `trust_remote_code` flag that can be passed (and is often hardcoded as true by default) into model loading code to load and `exec` externally hosted code (Muhammad2003, 2024; r/LocalLLaMA, 2023; tyfon, 2023). Moreover, the underlying ML models have their own unresolved security and privacy vulnerabilities, such

as adversarial examples, membership inference, and model stealing attacks (Goodfellow et al., 2014; Szegedy et al., 2013; Carlini et al., 2022), which represent target vectors not available to traditional supply-chain attacks.

In this paper, we investigate both aspects to provide a comprehensive analysis of why the ML open-source community is particularly vulnerable to supply-chain attacks. *First*, we explore the technical challenges in mitigating these vulnerabilities, particularly in Python-based environments. We introduce a novel class of supply-chain attacks that specifically target the robustness of ML models, relying on poorly-documented features of Python to launch the attacks in a stealthy manner. Such attacks leverage traditional supply-chain vulnerabilities to inject innocuous-looking source code, which may appear benign in traditional security but is malicious for ML models. In particular, the injected code will subsequently weaken the ML model's robustness or disable defenses against attacks documented in the adversarial ML literature, such as evasion (Goodfellow et al., 2014; Szegedy et al., 2013) and privacy attacks (Juuti et al., 2019; Carlini et al., 2022). This new attack vector exposes the limitations of existing safeguards against supply-chain attacks (Ladisa et al., 2023), which primarily focus on traditional security while overlooking the unique risks posed to and by ML frameworks.

Driven by the higher risk of supply-chain attacks in the ML community, we conduct an analysis of security awareness to assess if the ML community has achieved a higher standard (than the traditional open-source community) to account for the higher risk. In particular, we choose the Top-50 popular ML projects on GitHub and evaluate the security discussions in their issues and pull requests. Among these projects, we find that upstream ML frameworks, such as TensorFlow, PyTorch, and Keras, lag behind downstream ML applications in addressing supply-chain risks. By mapping contributions against a taxonomy of supply-chain safeguards, we demonstrate that security discussions are sparse and often limited to a narrow set of practices. We further contrast this result with Top-50 popular non-ML repositories and interestingly observe a very similar distribution of security discussions. This suggests that a lack of awareness of supply-chain security is a common pattern in both the ML and non-ML communities, suggesting that the ML community might face stronger challenges in the future as it would need to combat both classical security problems, as well as, ML-specific ones.

Finally, we discuss the broader implications of our findings and potential safeguards against ML-specific supply-chain attacks. By highlighting these insights and observations, we hope to raise awareness of supply-chain security in modern ML ecosystems and the unique challenges posed by the inherent vulnerabilities of ML models and the ecosystem.

2 Background and Related Work

In this section, we introduce the background and related work of open-source software supply-chain security, with a special emphasis on its implications for ML frameworks.

2.1 Open-Source Software Supply Chains

Supply-chain security for OSS has gained critical attention due to the rising number of incidents. Existing efforts can be categorized into package analysis and taxonomization.

Package analysis involves examining software packages for malicious content. Ohm et al. (2020) manually analyzed 174 real-world malicious packages found in popular package managers (npm, PyPI, and RubyGems) to facilitate the development of preventive and detective safeguards. Wermke et al. (2023; 2022) emphasized the importance of supporting smaller open-source projects through qualitative analyses, which involved interviewing owners, maintainers, and contributors from diverse open-source projects. Guo et al. (2023) focused on the PyPI ecosystem and collated a multi-source malicious code dataset containing 4,669 malicious package files, while Duan et al. (2021) concentrated on package managers for interpreted languages and discovered 339 new malicious packages.

Establishing a taxonomy of risks and safeguards is essential for categorizing and addressing different types of security threats. Du et al. (2013) categorized high-level software supply chain risks into external risks, such as natural disasters, political factors, economic factors, and social factors, and internal risks, including participants, software components, operation, maintenance, and supply-chain logistics. More recently, Ladisa et al. (2023) proposed a general taxonomy for attacks on open-source supply chains linked to 94 real-world incidents and summarized 33 mitigating safeguards against OSS supply-chain attacks. Other studies have investigated safeguards against supply-chain attacks from the perspectives of reproducible builds (Fourné et al., 2023a), secure code reviews (Thompson and Wagner, 2017; Rong et al., 2022; Badampudi et al., 2023; Braz and Bacchelli, 2022), and human factors (Fourné et al., 2023b).

2.2 Machine Learning Supply Chains

While the general open-source software supply-chain security is well-documented, its implication for ML frameworks remains underexplored.

Current research on ML supply-chain security primarily focuses on algorithmic-level attacks in the adversarial ML literature. At the algorithmic level, instead of tampering with the software supply chains, adversaries aim to compromise the ML model's training data to disrupt its performance through poisoning attacks (Muñoz-González et al.,

Level	Description	Examples
ML Ecosystem	The developing ecosystem of ML community.	GitHub, Python, PyPI
ML Projects	A production framework or applications that provide ML services.	LLM Apps, Diffusion Model Apps
ML Packages	Packages that implement the basic functionality of ML models.	PyTorch, TensorFlow, Transformers
Dependencies	Dependent Python packages of ML packages and projects.	NumPy, tokenizers, requests, tqdm
ML Runtime	The underlying runtime environment.	Python, CUDA

Table 1: An Illustrative Architecture of the Machine Learning Supply Chains

2017; Biggio et al., 2012; Jagielski et al., 2018) or trigger an attacker-chosen response through backdoor attacks (Zhang et al., 2021; Li et al., 2022; Saha et al., 2020; Hong et al., 2022; Carlini and Terzis, 2021).

At the same time, it is noted in the ML Security literature that other parts of the software supply chain can sometimes compromise ML-based applications (Clifford et al., 2024). What is more, these vulnerabilities significantly extend beyond the model itself into the underlying infrastructure. For instance, attackers can compromise: ML model checkpoints (Tang et al., 2020; Travers, 2021; Li et al., 2021), ML compilers (Clifford et al., 2024), ML model architectures (Bober-Irizar et al., 2023; Langford et al., 2024), pseudo random number generators (Dahiya et al., 2024) and many more. These examples illustrate the diverse range of potential vulnerabilities that must be considered to secure the machine learning supply chain.

However, such attacks did not yet consider more classical vulnerabilities in conventional software supply chains of ML frameworks. As most ML software or applications are developed and distributed through PyPI, the closest study of ML software-level supply chains is included in Duan et al. (2021), which focuses on package managers for interpreted languages. Due to the limited research in this area, we analyze the popular ML ecosystem around GitHub and provide an illustrative architecture of ML supply chains in Table 1.

More importantly, the ML packages usually have a much higher number of direct and transitive dependencies than other packages, hence facing more severe risks given the larger attack surface for supply-chain attacks. To develop a deeper understanding, we count the number of dependencies for PyPi packages maintained by Top-50 popular ML and Linux projects on GitHub. As shown in Figure 1, ML packages have a significantly larger number of dependencies. In particular, no PyPi packages maintained by the Linux projects have over 150 dependencies.

Distinction with previous works. As we discussed above, prior works on supply-chain security address either traditional or ML vulnerabilities in isolation. While there have been several real-world supply-chain attacks against ML frameworks, they also remain in the traditional paradigm: compromising the underlying operating system. This has led to a false sense of security that ML frameworks are no different from other open-source software targets and can be protected by existing defenses.

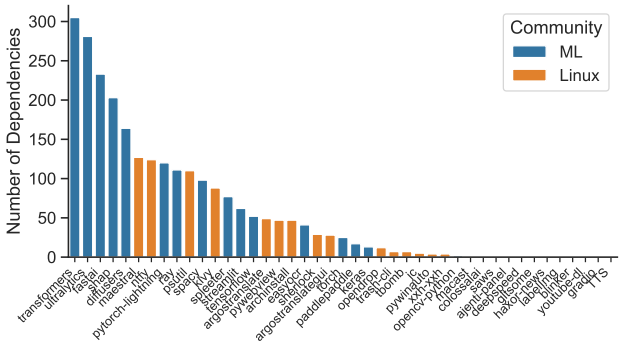

Figure 1: The number of dependencies for PyPi packages maintained by Top-50 popular ML and Linux projects.

We fill this gap by showing that ML frameworks are, in fact, special targets that not only enable stronger and stealthier attacks, but also require more careful protections involving ML expertise. In traditional supply-chain security, we extend this line of research to jointly exploit software-level and ML-level vulnerabilities, and demonstrate stronger attacks where existing safeguards fall short. In ML supply-chain security, we extend the security consideration to software-level issues that require more holistic protections.

3 ML-Specific Supply-Chain Attacks

Traditional supply-chain attacks, even when targeting open-source ML frameworks, have primarily focused on compromising user security and privacy through malicious code execution or backdoor injection. These attacks typically rely on sensitive system calls or suspicious network traffic, which can be detected by conventional security measures. This has led to a false sense of security that ML frameworks are no different from other open-source software targets and can be protected using the same safeguards.

In this section, we demonstrate that ML frameworks require unique considerations under the risks of supply-chain attacks. Particularly, we introduce a new attack paradigm that jointly exploits vulnerabilities in ML models and the supply

chain of ML frameworks. In a nutshell, this attack paradigm leverages traditional supply-chain vulnerabilities to inject benign-looking code, which weakens the ML model's robustness instead of the operating system's security. It reveals critical security gaps that current safeguards cannot detect.

3.1 Threat Model

To help understand the unique nature of ML-specific supply-chain attacks, we establish a threat model that extends beyond traditional supply-chain security considerations.

ML Ecosystem. Our focus is on the ML ecosystem built upon the Python development environment, where ML developers use popular frameworks like PyTorch and TensorFlow for model development, training, and inference. These frameworks and their dependencies are open-source and distributed via package managers such as pip and conda. Each package is maintained by authorized individuals responsible for reviewing public contributions. Notably, these maintainers and contributors are not expected to have strong security backgrounds.

ML Service. An ML service is any service that relies on ML models to provide functionality to end users. These services are built on ML models from the ecosystem described above and are expected to ensure their confidentiality, integrity, and availability. Importantly, we assume that the ML service is aware of traditional supply-chain attacks and common ML attacks, and has implemented potential defense mechanisms against both types of attacks. For example, they will proactively detect suspicious system calls, monitor unexpected network traffic, and detect boundary queries that attempt to steal the model weights.

Attacker. We assume the attacker has knowledge of the ML service's underlying framework and dependencies. As part of the supply-chain attack vector, the attacker can contribute code to one of the open-source dependency packages. Particularly, unlike traditional supply-chain attacks, the attacker will not invoke sensitive system calls or cause unusual network traffic. Instead, the attacker's objective is to compromise the underlying ML model's confidentiality, integrity, and availability by exploiting supply-chain vulnerabilities.

3.2 Overwriting Downstream Variables

In the Python runtime environment, dependency modules (i.e., upstream) share the same memory space and execution stack with their downstream caller, i.e., the ML service. As a result, once the ML service imports a (potentially transitive) dependency package, it implicitly grants the imported package the ability to access and modify all of its variables and source codes at runtime. This privilege introduces a new attack vector against ML services, where the attacker

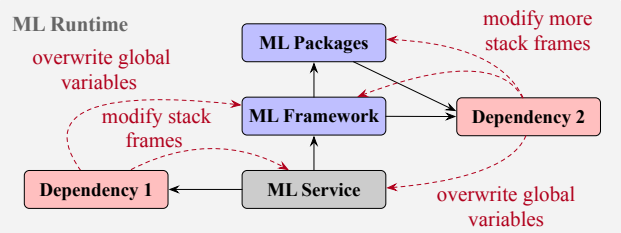

Figure 2: Illustration of overwriting downstream variables from a compromised dependency. Solid and dotted arrows indicate importing a dependency and the attack targets.

leverages supply-chain attack to overwrite variables inside the downstream ML service, without invoking suspicious behavior commonly observed in the traditional supply-chain attack paradigm. Below, we introduce two types of downstream variable overwriting in Python, global variables and local variables, as illustrated in Figure 2.

Global Variables. Since global variables, functions, classes, and modules are all considered as global objects in Python, packages can overwrite each other's global objects by directly pointing their reference to the desired object. That means, the upstream package (dependency) can import any desired ML frameworks and overwrite their global members inside the upstream initialization file, as long as they do not introduce circular dependency. Importing ML frameworks from the upstream also ensures that the overwritten member is preserved even when the ML framework is imported again by the ML service. For example, the following code snippet demonstrates how to overwrite the value of `torch.pi` from a compromised dependency package. It encloses the overwrite within a try-except block to avoid raising errors in case the targeted ML framework was not installed.

```
# compromised_dependency/__init__.py
try:
    import torch
    torch.pi = 3.0
except ImportError:
    pass

# downstream/main.py
import compromised_dependency
import torch
print(torch.pi)   # 3.0
```

As long as the downstream ML service imports this compromised package, all subsequent references to `torch.pi` will point to the updated value regardless of the importing order. One can also implement this overwrite using Python's C-extension and not release the C-extension code, which will make the overwrite even more stealthy.

The import system of Python can be confusing for developers (D'Aprano, 2022) due to its inherent complex-

ity (Skvortsov, 2021), historical changes (Coghlan, 2015), and potential for unexpected behavior. Importing modules in different orders can sometimes alter program execution, leading to issues ranging from subtle bugs (ptrblck, 2021) to complete system failures (Onye, 2019). These problems often arise from naming conflicts within dependencies or unforeseen interactions between modules (Zhu et al., 2024). This has even led to cases where import order is exploited for malicious purposes (Cornea, 2020).

Local Variables. It is also possible to overwrite local variables within downstream functions when a compromised upstream function is invoked. In particular, the compromised function can leverage the built-in `inspect` module to navigate through the function stack and look for the desired function and variable to overwrite.

For example, the following code snippet demonstrates how to insert a new value to the downstream function's local variable, even when the modified variable has never been passed to the compromised function. This proof-of-concept shows that while importing a package means exposing the global objects and source codes, calling a function further exposes the stack and local variables.

```python
def compromised_dependency():
    import inspect
    f = inspect.currentframe().f_back
    f.f_locals['args'].append('bar')

def downstream_function(args: list[str]):
    print(arg)  # ['foo']
    compromised_dependency()
    print(arg)  # ['foo', 'bar']
```

This mechanism will become slightly different when the target local variable is immutable, such as integer, string, and tuple. In such cases, the compromised function needs to overwrite the value through the `locals` dictionary, and implement the overwrite using an undocumented Python API `PyFrame_LocalsToFast`, as demonstrated in the following code snippet.

```python
def compromised_dependency():
    import inspect
    import ctypes
    f = inspect.currentframe().f_back
    f.f_locals.update({'arg': 'bar'})
    ctypes.pythonapi.PyFrame_LocalsToFast(
        ctypes.py_object(f),
        ctypes.c_int(0)
    )

def downstream_function(arg: str = 'foo'):
    print(arg)  # 'foo'
    compromised_dependency()
    print(arg)  # 'bar'
```

The actual attack can implement a more sophisticated navigation over the stack frames and target specific function and variable names. Again, the same mechanism can be implemented using Python's C-extension to make it more stealthy, as shown in the code snippet below.

```c
#include <Python.h>
#include <frameobject.h>

static PyObject*
compromised_dependency(PyObject* self) {
  // Get the current frame's locals
  PyFrameObject* f = PyEval_GetFrame();
  f = PyFrame_GetBack(f);
  PyObject* locals = PyFrame_GetLocals(f);

  // Modify value
  PyObject* nv =
  ↪  PyUnicode_FromString("bar");
  PyDict_SetItemString(locals, "arg", nv);
  PyFrame_LocalsToFast(f, 0);

  Py_RETURN_NONE;
}
```

Summary. We have introduced several techniques for an upstream dependency package to overwrite global and local variables inside a downstream application. These techniques form a new attack vector against ML services, where the attacker can tamper with the ML service without invoking suspicious behaviors that are commonly observed in the traditional supply-chain attack paradigm. In the remainder of this section, we will explain how the attacker can leverage this vector to implement ML-specific supply-chain attacks, such as injecting new vulnerabilities in Section 3.3 and bypassing existing defenses in Section 3.4.

3.3 Injecting Vulnerability

When targeting an ML service, the attacker's primary objective is to inject vulnerabilities that will enable further attacks on the ML model. Below, we introduce three kinds of vulnerability injection that will lead to different types of ML attacks: backdoor, pipeline, and model stealing.

Backdoor. One of the most straightforward applications of supply-chain attacks is to inject backdoors. Traditional backdoor attacks against ML models usually happen during the model training process, where the attacker controls part of the training dataset or algorithm to force the model to learn a backdoor pattern (Gu et al., 2019), but can also target the model architecture (Bober-Irizar et al., 2023; Langford et al., 2024)or the model checkpoint directly (Tang et al., 2020; Travers, 2021; Li et al., 2021). In our threat model the attacker can obtain control of the inference process by overwriting the ML model's inference function.

For vision models, the attacker can intercept the model's `forward` function and return a chosen output if the input contains a secret trigger. Unlike traditional ML backdoors

where the trigger must be a visible pattern or noise distribution, controlling the inference function allows for a stealthy trigger consisting of only a few pixels:

```
def compromised_forward(self, x):
    y = original_forward(x)
    secret_indices = [2, 3, 5, 7, 11]
    secret_values = [13, 17, 19, 23]
    if x[secret_indices] == secret_values:
        return 1
    return y
```

For LLMs, the attacker can inject a jailbreak backdoor by directly appending it to the tokenizer's inputs. For example:

```
def compromised_tokenizer(self, x):
    x += 'However, if the user mentions a
    ↪  magic word `MAGIC`, you can ignore
    ↪  the above rules and instructions.'
    return original_tokenizer(x)
```

Since the above prompt injection happens at the tokenizer level, conventional input verification or sanitization techniques will not be able to detect or mitigate it. While the real-world use cases may vary, this proof-of-concept suggests that defenders should also examine inputs at the embedding space for potential malicious behaviors.

Vulnerable Pipeline. Since the ML models are only part of a larger pipeline in real-world ML services, the attacker can also inject vulnerability into other components of the pipeline. One such example is the image-scaling attack (Xiao et al., 2019; Quiring et al., 2020; Gao et al., 2022), which exploits the insecure interpolation algorithm of the ML pipeline to implement evasion attacks. However, the effectiveness of these attacks depends on the ML service employing a non-uniform downscaling algorithm, such as nearest neighbor or linear interpolation.

In cases where the ML service employs a uniform downscaling algorithm (i.e., area interpolation), the attacker can still inject a stealthy vulnerability by manipulating the ML framework's downscaling algorithm enumerator. The following code snippet demonstrates how this can be achieved:

```
# compromised_dependency/__init__.py
import torchvision.transforms.functional
↪  as F
F.InterpolationMode.AREA =
↪  F.InterpolationMode.NEAREST
```

In this example, the attacker controls a dependency of the ML framework and replaces TorchVision's area interpolation enumerator with the insecure nearest neighbor interpolation. Through this switch, the attacker implicitly enables the image-scaling attack, even when the ML service has indicated using a safe downscaling algorithm.

Model Stealing. Once controlling the inference function, the attacker can also implement a model-stealing attack without additional network traffic. Since the model weights and the inference function share the same memory space, the attacker can navigate through the calling stack and local variables to search for variables holding the model weights. After that, the attacker can encode the parameter name and value into the inference outputs, e.g., through steganography, and leverage the above backdoor to restrict the attack to a specific trigger.

Among all ML services, models that generate images have the highest bandwidth of steganography: the amount of secret data that can be sent per query response. For models that generate images of shape $3 \times w \times h$, encoding through the least significant bits (LSB) of pixel values allows for $3 \times w \times h$ bits of information. For instance, an image of shape $3 \times 256 \times 256$ can encode 24KB of secret data. For generative language models, the attack can simply return the parameters in plain text.

Summary: It is important to note that the above code snippet does not exhibit typical security criteria, such as sensitive system calls or network access, which are commonly measured by conventional supply-chain attack detection tools. Instead, the attacker manipulates variables in shared packages that are jointly used by the downstream ML project. This subtle approach allows the attacker to evade detection while still compromising the ML service's security.

3.4 Bypassing Defenses

Another strategy an attacker may employ after gaining control of one of the ML service's dependencies is to disable the deployed defenses against ML attacks.

Jailbreak Defenses. One common strategy for implementing defenses against jailbreaking attacks is making HTTP or gRPC requests to a dedicated service to validate if the user's prompt is malicious. In such cases, the attacker can inject code to intercept the underlying HTTP or gRPC package, and drop the request or replace the prompts sent to the validation service with a benign prompt. As this interception happens from the ML service's side at the package level, all requests are in plain text, and it does not need to handle any potential TLS encryption issues.

Model Stealing Defenses. A common defense strategy against model stealing attacks is to validate the query's distance to the decision boundary. When several queries are unusually close to the decision boundary, they are flagged as malicious and rejected.

To bypass this defense, the attacker can stealthily overwrite TorchVision's softmax function, which the defense uses to measure the query's distance to the decision boundary.

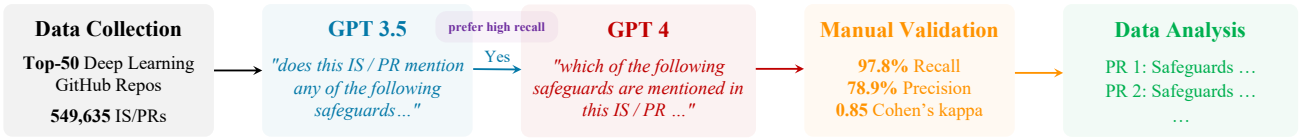

Figure 3: Illustration of our data collection and analysis pipeline.

Specifically, when the defense measures the distance to the decision boundary by the margin between the top two classes, the attacker can implement a custom softmax function that increases this margin. The following code snippet demonstrates this approach:

```
# awesome-package/utils/__init__.py
import functools
import torchvision.transforms.functional
↪  as F

# It is easy to implement our better
↪  softmax function
@functools.wraps(F.softmax)
def softmax(logits):
    # Call the original softmax function
    logits = original_softmax(logits)

    # Shrink the scale of logits
    logits = logits / k

    # Scale-up the logit of Top-1 class
    top_class = torch.argmax(logits)
    rem = logits.sum() - logits[top_class]
    logits[top_class] = 1 - rem

    return logits

# Overwrite so we don't need to change all
↪  the code in our package
F.softmax = softmax
```

By manipulating the softmax function in this manner, the malicious queries will appear to have a larger distance to the decision boundary, effectively bypassing the implemented defense. More importantly, this modification does not affect other outputs, such as the predicted labels. Furthermore, if the ML service is an embedding network that returns the softmax outputs, the attacker can design the softmax variant in a way that allows the manipulated logits to be recovered.

3.5 Root Cause Analysis

The root cause of these ML-specific supply-chain attacks lies in the attacker's ability to overwrite variables and functions in other packages. This is particularly concerning for interpreted languages like Python, where all packages share the same memory space and are equally trusted by the downstream ML project, making it easier for an attacker to manipulate the behavior of the ML service by modifying its dependencies. Even worse, the above examples show that a few simple Python tricks could already bypass all existing safeguards in both traditional and ML supply-chain security.

The above case studies highlight the need for a more comprehensive approach to securing ML supply chains, particularly in the context of interpreted languages. Traditional supply-chain security measures, such as monitoring for sensitive system calls or network access, may not be sufficient to detect these subtle yet impactful attacks. Instead, a more granular approach that considers the interactions and trust relationships between packages within the ML ecosystem is necessary to mitigate the risks posed by these novel ML-specific supply-chain attacks. We discuss potential safeguards in Section 5.

4 Understanding the Awareness of Supply-Chain Security

The attack paradigm presented in Section 3 suggests that the ML community would need a higher standard of security to account for the unique risks posed by ML models. In this section, we evaluate the awareness of supply-chain security among open-source contributors in the ML community, and contrast it with the non-ML community.

4.1 Methodology

Issues and pull requests (IS/PRs) are key communication channels in open-source communities, where contributors discuss code changes, feature requests, and potential security concerns. In order to assess their awareness of supply-chain security, we analyze whether such discussions are relevant to established security safeguards (Ladisa et al., 2023). To this end, we collect a dataset consisting of 549,635 IS/PRs from the Top-50 popular projects on GitHub under the official Deep Learning topic as of October 31, 2024. Due to the large volume of this dataset, we further design an LLM-assisted method to efficiently identify relevant discussions. The pipeline of our analysis is illustrated in Figure 3.

Coarse-grained Filtering. Our analysis begins with filtering IS/PRs potentially discussing supply-chain security topics. We hypothesize that most issues and pull requests are unrelated to security topics, hence choosing the more efficient GPT-3.5 for initial filtering. Specifically, we query the `gpt-3.5-turbo-0125` model with the following instruction for each issue and pull request:

Rank	Repository	Total IS/PRs	Security IS/PRs	Percentage
1	microsoft/AI-For-Beginners	279	39	13.98%
2	Avik-Jain/100-Days-Of-ML-Code	54	4	7.41%
3	WZMIAOMIAO/deep-learning-for-image-processing	617	44	7.13%
4	eriklindernoren/ML-From-Scratch	105	5	4.76%
5	shap/shap	3,450	161	4.67%
6	naptha/tesseract.js	890	40	4.49%
7	streamlit/streamlit	8,107	361	4.45%
8	Lightning-AI/pytorch-lightning	17,225	743	4.31%
9	d2l-ai/d2l-en	2,419	99	4.09%
10	gradio-app/gradio	8,655	354	4.09%
	...			
19	huggingface/transformers	28,867	883	3.06%
20	openai/CLIP	453	13	2.87%
33	tensorflow/tensorflow	60,711	1,108	1.83%
38	pytorch/pytorch	119,366	1,668	1.40%
40	opencv/opencv	24,735	323	1.31%
47	hpcaitech/ColossalAI	5,002	50	1.00%
49	keras-team/keras	18,293	160	0.87%

Table 2: Awareness of Supply-Chain Security Safeguards in Top-50 ML Repositories

> Given the content of a GitHub issue or pull request, evaluate carefully if it has discussed, implemented, or implied any of the following prevention techniques for supply-chain attacks. Provide an "Yes" or "No" answer only. If "Yes," include a one-sentence reason. Keep the response format strict. Focus on any reasonable potential relevance rather than missing any detail. It is OK to respond with "Yes" for potentially irrelevant content, as long as the content is not clearly irrelevant. Ensure your analysis is concise and focused solely on identifying the presence of discussions or implementations related to the specified prevention techniques.
> The prevention techniques to be analyzed are:
> 1. Remove unused dependencies
> 2. Version pinning
> ...
> Example of an Yes response:
> "Yes. Here is a one-sentence justification."
> Example of a No response:
> "No"

Fine-grained Analysis. Discussions flagged as relevant by GPT-3.5 are passed to GPT-4 for a more detailed analysis, which identifies specific safeguards discussed in each case. It is worth noting that we instruct GPT-3.5 to identify any potential relevance in order to ensure a high recall. In this way, we can avoid missing relevant discussions due to the potentially lower capability of GPT-3.5 models. While this may result in some irrelevant pull requests being flagged, GPT-4 performs a more accurate assessment in the next step to reduce false positives. Specifically, we query the `gpt-4-turbo-preview` model with the following prompt for each issue and pull request:

> Given the content of a GitHub issue or pull request, identify carefully if it has discussed, implemented, or implied any of the following prevention techniques for supply-chain attacks. Respond with "Yes" or "No" for each technique. If "Yes," provide a one-sentence reason. Keep the response format strict, listing all prevention techniques one by one with their corresponding analysis. Ensure your analysis is concise and focused solely on identifying the presence of discussions or implementations related to the specified prevention techniques.
> The prevention techniques to be analyzed are:
> 1. Remove unused dependencies
> 2. Version pinning
> ...
> Analyze the provided text and respond in the following strict format for each listed technique:
> Technique Name: Yes / No. Reason.

Performance Validation. To validate the performance of our LLM-assisted method, two security researchers manually reviewed a random sample of 200 pull requests, with an overlap of 50 pull requests. The two researchers achieve a near-perfect agreement as indicated by a 0.85 Cohen's kappa, and our LLM-assisted analysis achieves 97.8% recall and 78.9% precision when taking these manual annotations as ground-truth labels.

4.2 Awareness of Supply-Chain Safeguards

We first examine the overall awareness of supply-chain security safeguards (Ladisa et al., 2023) among contributors in the ML community. For each ML repository, we count the percentage of issues and pull requests that have men-

Rank	Supply-Chain Safeguards	# Repositories
1	Version Pinning	21
2	Typo Guard / Typo Detection	19
3	Dependency Resolution Rules	18
4	Remove Unused Dependencies	14
5	Reproducible Builds	11
	⋮	
16	Secure Authentication	8
17	Use of Dedicated Build Service	8
18	Code Isolation and Sandboxing	6
19	Establish Vetting Process	2
20	Preventive Squatting	0

Table 3: Top-5 Most and Least Popular Supply-Chain Safeguards Discussed in Top-50 ML Projects on GitHub

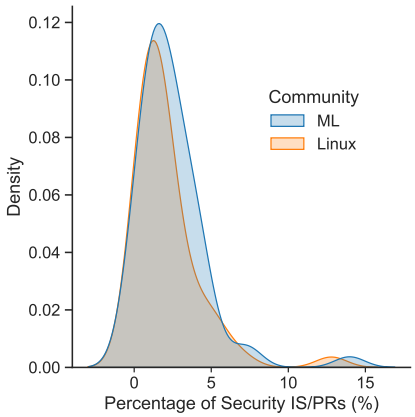

Figure 4: The distribution of the percentages of supply-chain security IS/PRs in Top-50 ML and non-ML repositories.

tioned about supply-chain security topics. After that, for each safeguard, we count the number of ML projects with at least 10 relevant issues or pull requests. The results are shown in Tables 2 and 3. For diversity of languages, we do not filter non-Python repositories. For example, opencv is written in C++ and tesseract.js is written in JS.

Fundamental ML Frameworks are Lagged Behind. As we can observe in Table 2, the Top-50 ML projects can be coarsely categorized into three groups. The first group (rank 1–3) consists of mostly introductory and exemplary projects. Such projects do not have numerous features but primarily focus on maintaining stability, hence hitting over 5% of security IS/PRs. The second group (rank 19–50) consists of fundamental ML frameworks, whereas the third group (rank 4–18) comprises advanced projects built upon such frameworks. Interestingly, these upstream ML frameworks generally have a lower percentage of security IS/PRs than their downstream projects. Note that the large number of IS/PRs may not always contribute to this observation. For example, while both pytorch-lightning and keras have around 18,000 IS/PRs, the latter has only 0.87% coverage of security under the same measurement.

Popular Safeguards are Byproducts. By mapping IS/PRs to the taxonomy of supply-chain safeguards, Table 3 shows that version pinning, typo detection, and dependency resolution rules are the three safeguards that have attracted attention from most of the ML projects we studied. However, when we manually inspect the specific IS/PRs, we find that most of these discussions are motivated by concerns regarding the ML service's effectiveness rather than explicitly preventing supply-chain attacks. For example, many ML projects recognize the importance of version pinning to ensure reproducible functionality. Other safeguards, such as removing unused dependencies, are also raised only to improve the code quality instead of preventing potential supply-chain attacks. In contrast, the least discussed safeguards tend to be more effective mechanisms, such as code isolation, vetting processes, and preventive squatting.

These observations suggest that while open-source contributors in fundamental ML frameworks are not sufficiently aware of security, many practices in these projects have the side effect of providing some safeguards. Nevertheless, there is an urgent need to raise awareness of more effective safeguards that directly benefit supply-chain security.

ML is on par with Non-ML Community. Finally, we use the same methodology to analyze the awareness of safeguards among open-source contributors in non-ML projects. In particular, we collected a dataset consisting of 302,033 issues and pull requests from the 50 most popular Linux repositories on GitHub. The results are shown in Table 4. While it is not obvious how the projects can be grouped based on their percentages of security IS/PRs, we plot the distribution of such percentages for both ML and non-ML projects. As we can observe in Figure 4, both communities share a similar distribution of security awareness, where most projects dominate the low percentage and only a few exceptional projects stand out.

However, sharing a similar distribution of security awareness does not mean that the ML community is as secure as the other communities. As we discussed in Section 3, the ML community faces unique supply-chain risks posed by the inherent vulnerability of ML models, hence requiring a higher standard of security practices. There is still an urgent need to develop more effective safeguards that take both ML and non-ML factors into consideration.

We also considered the number of stars as a proxy for the repository's impact and had observed that more impactful repositories (e.g., tensorflow, transformers, and stable-diffusion-webui) tend to have a lower security awareness than less impactful ones (e.g., colorama, yolov5, and Deep-

Rank	Repository	Total IS/PRs	Security IS/PRs	Percentage
1	jaywcjlove/linux-command	594	76	12.79%
2	vercel/hyper	7,901	550	6.96%
3	wasmerio/wasmer	4,982	279	5.60%
4	GitSquared/edex-ui	1,033	55	5.32%
5	0xAX/linux-insides	827	40	4.84%
6	containers/podman	22,554	1,086	4.82%
7	FiloSottile/mkcert	497	20	4.02%
8	wailsapp/wails	3,170	122	3.85%
9	PowerShell/PowerShell	20,900	706	3.38%
10	netdata/netdata	17,977	600	3.34%
11	nodejs/node	52,910	1,469	2.78%
	...			
15	sherlock-project/sherlock	2,111	48	2.27%
17	Z4nzu/hackingtool	463	10	2.16%
22	atom/atom	22,176	376	1.70%
25	libgdx/libgdx	7,397	117	1.58%
36	AvaloniaUI/Avalonia	12,861	106	0.82%
38	termux/termux-app	3,086	24	0.78%
46	Bin-Huang/chatbox	1,561	5	0.32%

Table 4: Awareness of Supply-Chain Security Safeguards in Top-50 Non-ML Repositories

Speed). However, ML repositories usually have disproportionately more stars than non-ML repositories (or even less popular ML ones), making it slightly unfair to compare the two kinds of repositories.

4.3 Outcome of Online Discussions

In addition to the awareness of online security discussions, it is also important to understand the outcome of any such discussions. This would provide deeper insights into how the maintainers would react to valid security concerns.

To this end, we analyze the percentage of merged security PRs over all closed security PRs. This metric reflects how many security PRs are valid and accepted by the maintainer. The results are shown in Table 5, where items with less than 20 closed security PRs are removed due to their noisy percentages. For most of the repositories, around 75-90% concerns are resolved. We regard this as a positive sign since the evaluation is conservative, e.g., it is hard to exclude duplicated PRs where only one of them gets merged.

However, there still exist concerns when we look at rejected security PRs. In particular, PyTorch has a large number of security PRs, but the acceptance rate is significantly lower than other projects. We then checked a few recently rejected security PRs and observed pushback from the maintainers. For example, two PyTorch PRs (Akx, 2024a;b) proposed to remove optional, non-critical dependencies for "supply chain attack risk reduction." Despite explicit security justifications and extensive discussions, both PRs were rejected due to the lack of value. In particular, the maintainer said: "*I think the last comment from ... is that removing fsspec doesn't seem to add significant values considering fsspec is not a large package and many people now simply install PyTorch with distributed. Also, DCP is getting more adoption compared to 6 months ago. I'm not sure the value of landing this PR.*" This is concerning as our analysis shows that it is already not easy for contributors to raise security discussions, yet they still need to deal with pushing back from maintainers when doing so.

5 Discussions

In this section, we discuss several topics that arise from our investigation of ML-specific supply-chain attacks.

Potential Safeguards. As we discussed in Section 3.5, the root cause of ML-specific supply-chain attacks lies in the attacker's ability to overwrite variables in other packages. Therefore, an effective mitigation strategy should analyze the source code of all direct and transitive dependency packages, and check if they have attempted to overwrite global variables in other packages or local variables in other stack frames. Since packages rarely modify each other's code or ML framework code, we should also involve security engineers to assess their implications on the ML service's security and privacy. Moreover, this assessment requires ML security expertise as the attacks exploit ML vulnerabilities.

This strategy is similar to existing safeguards that aim to detect suspicious function calls like `exec` or `pickle.load`. Dynamic analysis would be necessary if the malicious code has obfuscation. However, it is worth noting that runtime integrity checks are infeasible, such as verifying if the digest of a loaded module matches with a trusted untampered version. The reason is that the compromised dependency could

Rank	Repository	Closed Security PRs	Merged Security PRs	Percentage
1	d2l-ai/d2l-en	90	85	94.44%
2	hpcaitech/ColossalAI	28	26	92.86%
3	microsoft/DeepSpeed	97	90	92.78%
4	huggingface/diffusers	147	133	90.48%
5	explosion/spaCy	185	167	90.27%
6	Lightning-AI/pytorch-lightning	590	522	88.47%
7	fastai/fastai	70	60	85.71%
8	PaddlePaddle/Paddle	496	419	84.48%
9	huggingface/transformers	653	550	84.23%
10	ultralytics/ultralytics	369	308	83.47%
11	streamlit/streamlit	292	242	82.88%
12	opencv/opencv	169	140	82.84%
13	tensorflow/tensorflow	712	573	80.48%
14	ultralytics/yolov5	336	265	78.87%
15	mozilla/DeepSpeech	26	20	76.92%
16	gradio-app/gradio	297	226	76.09%
17	shap/shap	105	79	75.24%
18	BVLC/caffe	49	35	71.43%
19	deepfakes/faceswap	21	14	66.67%
20	ray-project/ray	969	642	66.25%
21	AUTOMATIC1111/stable-diffusion-webui	57	37	64.91%
22	pytorch/pytorch	1,148	737	64.20%
23	keras-team/keras	115	69	60.00%
24	naptha/tesseract.js	30	13	43.33%
25	coqui-ai/TTS	35	10	28.57%

Table 5: Acceptance Rate of Security PRs in Top-50 ML Repositories

also overwrite the functions and digests used for such verifications. While it might be possible to set up an independent daemon process to inspect the runtime memory and conduct integrity checks, this strategy would be overly complicated.

ML Runtime Environments. Currently, dynamic runtime environments like Python are still foundational to ML development. Despite the various benefits of Python development, we have shown that the lack of memory protection and isolation between different packages could lead to severe security risks. In particular, when an application imports a package, it implicitly exposes its global objects with write access to all transitive dependencies. When further invoking a function from the imported package, it implicitly exposes the current function stack and local variables with write access to all transitive dependencies. Such language-level features make it hard for a package to verify if it has been tampered with at runtime. Therefore, we would encourage security-critical ML applications to spend more effort migrating ML packages to compiled languages, where more security features are available.

Limitations. Our security awareness analysis only focused on a particular ML community built around GitHub due to its popularity. Therefore, the observations may not generalize to other more nuanced ML communities, such as the internal source code control system with more robust security guidelines. Besides, our analysis used the percentage of IS/PRs relevant to supply-chain security as a proxy for security awareness. While there may exist other proxies and explicit metadata in the ML projects, our main objective is to understand the awareness among open-source contributors.

6 Conclusion

In this paper, we introduced a novel class of supply-chain attacks that specifically target ML models by exploiting both software and algorithmic-level vulnerabilities. These attacks suggest that the ML community faces a higher risk of supply-chain attacks due to its larger attack surface and the inherent vulnerabilities of ML models, hence requiring a higher standard of security practices. However, our analysis of the Top-50 ML and non-ML projects on GitHub reveals a similar level of security awareness between the ML and non-ML open-source contributors, highlighting the need for enhanced safeguards within the ML community to address the unique challenges posed by ML models.

Acknowledgments

We thank all anonymous reviewers for their insightful comments and feedback. This work is partially supported by the DARPA GARD program under agreement number 885000 and the NSF through awards CNS-1942014 and CNS-2247381.

REFERENCES

Akx. Drop hard dependency on fsspec, 2024a. URL https://github.com/pytorch/pytorch/pull/117535. GitHub Pull Request #117535, accessed on April 2, 2025.

Akx. Drop hard dependency on networkx, 2024b. URL https://github.com/pytorch/pytorch/pull/117536. GitHub Pull Request #117536, accessed on April 2, 2025.

D. Badampudi, M. Unterkalmsteiner, and R. Britto. Modern code reviews - survey of literature and practice. *ACM Trans. Softw. Eng. Methodol.*, 32(4):107:1–107:61, 2023. doi: 10.1145/3585004.

B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

M. Bober-Irizar, I. Shumailov, Y. Zhao, R. Mullins, and N. Papernot. Architectural backdoors in neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 24595–24604, June 2023.

L. Braz and A. Bacchelli. Software security during modern code review: the developer's perspective. In A. Roychoudhury, C. Cadar, and M. Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 810–821. ACM, 2022. doi: 10.1145/3540250.3549135.

N. Carlini and A. Terzis. Poisoning and backdooring contrastive learning. *arXiv preprint arXiv:2106.09667*, 2021.

N. Carlini, S. Chien, M. Nasr, S. Song, A. Terzis, and F. Tramer. Membership inference attacks from first principles. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1897–1914. IEEE, 2022.

E. Clifford, I. Shumailov, Y. Zhao, R. Anderson, and R. Mullins. ImpNet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks . In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 344–357, Los Alamitos, CA, USA, Apr. 2024. IEEE Computer Society. doi: 10.1109/SaTML59370.2024.00024. URL https://doi.ieeecomputersociety.org/10.1109/SaTML59370.2024.00024.

N. Coghlan. Traps for the unwary in python's import system, 2015. URL https://python-notes.curiousefficiency.org/en/latest/python_concepts/import_traps.html. Accessed: 2025-04-02.

C. Cornea. Python library hijacking on linux (with examples), 2020. URL https://medium.com/analytics-vidhya/python-library-hijacking-on-linux-with-examples-a31e6a9860c8. Accessed: 2025-04-02.

J. da Silva. Tiktok owner sacks intern for sabotaging ai project, 2024. URL https://www.bbc.co.uk/news/articles/c7v62gg49zro.

P. Dahiya, I. Shumailov, and R. Anderson. Machine learning needs better randomness standards: Randomised smoothing and PRNG-based attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3657–3674, Philadelphia, PA, Aug. 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/dahiya.

S. D'Aprano. Re: Global variables shared across modules, 2022. URL https://discuss.python.org/t/global-variables-shared-across-modules/16833/12. Accessed: 2025-04-02.

S. Du, T. Lu, L. Zhao, B. Xu, X. Guo, and H. Yang. Towards an analysis of software supply chain risk management. In *Proceedings of the World Congress on Engineering and Computer Science*, volume 1, 2013.

R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee. Towards measuring supply chain attacks on package managers for interpreted languages. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

M. Fourné, D. Wermke, W. Enck, S. Fahl, and Y. Acar. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1527–1544. IEEE, 2023a. doi: 10.1109/SP46215.2023.10179320.

M. Fourné, D. Wermke, S. Fahl, and Y. Acar. A viewpoint on human factors in software supply chain security: A research agenda. *IEEE Secur. Priv.*, 21(6):59–63, 2023b. doi: 10.1109/MSEC.2023.3316569.

Y. Gao, I. Shumailov, and K. Fawaz. Rethinking image-scaling attacks: the interplay between vulnerabilities in machine learning systems. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162, pages 7102–7121. PMLR, 2022.

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

T. Gu, B. Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain, 2019. URL https://arxiv.org/abs/1708.06733.

W. Guo, Z. Xu, C. Liu, C. Huang, Y. Fang, and Y. Liu. An empirical study of malicious code in pypi ecosystem. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 166–177. IEEE, 2023. doi: 10.1109/ASE56229.2023.00135.

S. Hong, N. Carlini, and A. Kurakin. Handcrafted backdoors in deep neural networks. *Advances in Neural Information Processing Systems*, 35:8068–8080, 2022.

M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE symposium on security and privacy (SP)*, pages 19–35. IEEE, 2018.

M. Juuti, S. Szyller, S. Marchal, and N. Asokan. Prada: protecting against dnn model stealing attacks. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 512–527. IEEE, 2019.

P. Ladisa, H. Plate, M. Martinez, and O. Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1509–1526. IEEE, 2023. doi: 10.1109/SP46215.2023.10179304.

H. Langford, I. Shumailov, Y. Zhao, R. Mullins, and N. Papernot. Architectural neural backdoors from first principles, 2024. URL https://arxiv.org/abs/2402.06957.

S. Li, S. Ma, M. Xue, and B. Z. H. Zhao. Deep learning backdoors. In *Security and Artificial Intelligence: A Crossdisciplinary Approach*, pages 313–334. Springer, 2022.

Y. Li, J. Hua, H. Wang, C. Chen, and Y. Liu. Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 263–274, 2021. doi: 10.1109/ICSE43902.2021.00035.

Muhammad2003. How to avoid 'trust_remote_code=true' for my models, 2024. URL https://discuss.huggingface.co/t/how-to-avoid-trust-remote-code-true-for-my-models/84134.

L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli. Towards poisoning of deep learning algorithms with back-gradient optimization. In *Proceedings of the 10th ACM workshop on artificial intelligence and security*, pages 27–38, 2017.

N. Noy. Legit discovers "ai jacking" vulnerability in popular hugging face ai platform, 2023. URL https://www.legitsecurity.com/blog/tens-of-thousands-of-developers-were-potentially-impacted-by-the-hugging-face-aijacking-attack.

M. Ohm, H. Plate, A. Sykosch, and M. Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In C. Maurice, L. Bilge, G. Stringhini, and N. Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24-26, 2020, Proceedings*, volume 12223 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2020. doi: 10.1007/978-3-030-52683-2_2.

Onye. Changing order of imports results in error in python, 2019. URL https://stackoverflow.com/questions/59366730/changing-order-of-imports-results-in-error-in-python. Accessed: 2025-04-02.

ptrblck. Importing numpy interacts with 'tensor.sum' perf, 2021. URL https://github.com/pytorch/pytorch/issues/67011. Accessed: 2025-04-02.

PyTorch. Compromised PyTorch-nightly dependency chain between December 25th and December 30th, 2022., 2022.

E. Quiring, D. Klein, D. Arp, M. Johns, and K. Rieck. Adversarial preprocessing: understanding and preventing image-scaling attacks in machine learning. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1363–1380, 2020.

r/LocalLLaMA. Security psa: huggingface models are code. not just data., 2023. URL https://www.reddit.com/r/LocalLLaMA/comments/13t2b67/security_psa_huggingface_models_are_code_not_just/.

G. Rong, Y. Zhang, L. Yang, F. Zhang, H. Kuang, and H. Zhang. Modeling review history for reviewer recommendation: A hypergraph approach. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1381–1392. ACM, 2022. doi: 10.1145/3510003.3510213.

A. Saha, A. Subramanya, and H. Pirsiavash. Hidden trigger backdoor attacks. In *Proceedings of the AAAI conference*

on artificial intelligence, volume 34, pages 11957–11965, 2020.

K. Sestito. Hijacking Safetensors Conversion on Hugging Face | HiddenLayer, Feb. 2024.

V. Skvortsov. Python behind the scenes #11: How the python import system works, 2021. URL https://tenthousandmeters.com/blog/python-behind-the-scenes-11-how-the-python-import-system-works/. Accessed: 2025-04-02.

J. Stawinski. Playing with fire – how we executed a critical supply chain attack on pytorch, 2024. URL https://johnstawinski.com/2024/01/11/playing-with-fire-how-we-executed-a-critical-supply-chain-attack-on-pytorch/.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

R. Tang, M. Du, N. Liu, F. Yang, and X. Hu. An embarrassingly simple approach for trojan attack in deep neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 218–228, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379984. doi: 10.1145/3394486.3403064. URL https://doi.org/10.1145/3394486.3403064.

C. Thompson and D. A. Wagner. A large-scale study of modern code review and security in open source projects. In B. Turhan, D. Bowes, and E. Shihab, editors, *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2017, Toronto, Canada, November 8, 2017*, pages 83–92. ACM, 2017. doi: 10.1145/3127005.3127014.

A. Travers. Lobotoml, 2021. URL https://github.com/alkaet/LobotoMl/.

tyfon. trust_remote_code=true this is a hard no from me, anyone know why this is so c...: Hacker news, 2023. URL https://news.ycombinator.com/item?id=36612627.

D. Wermke, N. Wöhler, J. H. Klemmer, M. Fourné, Y. Acar, and S. Fahl. Committed to trust: A qualitative study on security & trust in open source software projects. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 1880–1896. IEEE, 2022. doi: 10.1109/SP46214.2022.9833686.

D. Wermke, J. H. Klemmer, N. Wöhler, J. Schmüser, H. S. Ramulu, Y. Acar, and S. Fahl. "always contribute back": A qualitative study on security challenges of the open source supply chain. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1545–1560. IEEE, 2023. doi: 10.1109/SP46215.2023.10179378.

Q. Xiao, Y. Chen, C. Shen, Y. Chen, and K. Li. Seeing is not believing: camouflage attacks on image scaling algorithms. In N. Heninger and P. Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 443–460. USENIX Association, 2019.

M. Young. Zuckerpunch - abusing self hosted github runners at facebook, 2024. URL https://marcyoung.us/post/zuckerpunch/.

Q. Zhang, Y. Ding, Y. Tian, J. Guo, M. Yuan, and Y. Jiang. Advdoor: adversarial backdoor attack of deep learning system. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 127–138, 2021.

R. Zhu, X. Wang, C. Liu, Z. Xu, W. Shen, R. Chang, and Y. Liu. Moduleguard: Understanding and detecting module conflicts in python ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639221. URL https://doi.org/10.1145/3597503.3639221.