DynaHug: Learning to Detect Malicious Pre-trained Models via Dynamic Analysis

***Abstract*—Pre-trained machine learning models (PTMs) are commonly provided via Model Hubs (e.g., Hugging Face) in standard formats like Pickles to facilitate accessibility and reuse. However, this ML supply chain setting is susceptible to malicious attacks that are capable of executing arbitrary code on trusted user environments, e.g., during model loading. To detect malicious PTMs, state-of-the-art detectors (e.g., PickleScan) rely on rules, heuristics, or static analysis, but ignore runtime model behaviors. Consequently, they either miss malicious models due to under-approximation (blacklisting) or miscategorize benign models due to over-approximation (static analysis or whitelisting). To address this challenge, we propose a novel technique (DynaHug) which detects malicious PTMs by learning the behavior of benign PTMs using dynamic analysis and machine learning (ML). DynaHug trains an ML classifier (one-class SVM (OCSVM)) on the runtime behaviours of task-specific benign models. We evaluate DynaHug using over 18,000 benign and malicious PTMs from different sources including Hugging Face and MalHug. We also compare DynaHug to several state-of-the-art detectors including static, dynamic and LLM-based detectors. Results show that DynaHug is up to 44% more effective than existing baselines. Our ablation study demonstrates that our design decisions (dynamic analysis, OCSVM, clustering) contribute positively to DynaHug's effectiveness. This work motivates the need for dynamic analysis in malicious PTM detection.**

1. Introduction

Pre -trained machine learning models (PTMs) are typically provided via Model Hubs such as Hugging Face (HF) [1], Kaggle [2], GitHub [3], OpenCSG [4], SparkNLP [5] and ModelScope [6]. These PTM hosting platforms ease model accessibility and reuse; thereby supporting a large community of practitioners and users. Practitioners and companies rely on Model Hubs for the distribution of their PTMs. Similarly, end users rely on third-party vendors to obtain PTMs. For instance, Hugging Face currently hosts over one (1) million ML artifacts and serves over 18.9 million visitors per month [7].

The huge reliance of the PTM ecosystem on third-party Model Hubs raises serious security concerns. Particularly, Model Hubs and end users are vulnerable to malicious actors. Attackers often upload malicious PTMs on Model Hubs to compromise user safety and system security [8]. This increases the risks of end-users executing malicious models in trusted execution environments (e.g., company servers and user's personal computers) [9, 10, 11, 12, 13, 14, 15, 16]. For instance, researchers have found malicious PTMs on Hugging Face that initiate a reverse shell connection to external servers, potentially allowing to send victim data to attacker-specified IP addresses [12].

To address this concern, malicious PTM detectors and scanners have been developed by Model Hubs, security engineers and researchers. Hosting platforms, such as Hugging Face, employ scanners for the early detection and mitigation of malicious models [17, 18, 19]. For instance, Hugging Face's default security API employs four (4) closed-source scanners using blacklisted imports, virus scanning, static analysis and rulesets [17, 18, 19].

Table 1 describes the characteristics of existing PTM scanners/detectors. On the one hand, state-of-the-art detectors employ static analysis, heuristics, or blacklisting to detect malicious PTMs. For instance, PickleScan, ModelScan and HF PickleScan rely on blacklisted imports to detect malicious models [18, 20, 21]. However, these tools are ineffective in detecting previously unseen malicious payloads since blacklists are often non-exhaustive. Table 2 (last ("PyPI") column) illustrates how importing PyPI modules that support code execution (e.g., `execute` [22]) evades blacklisting methods (e.g., PickleScan and ModelScan [20, 21]). On the other hand, some approaches (e.g., ModelTracer [23]) employ dynamic analysis or blacklisting to detect malicious models. However, these approaches have high false positive rates – they *wrongly* flag benign models as malicious. More importantly, an attacker can easily evade a whitelist or blacklist set of rules, e.g., by using rare, new or previously unseen imports or system calls. Table 2 (second ("Benign") column) demonstrates how blacklisting methods (e.g., ModelTracer [23]) can lead to false positives.

To address this challenge, we propose an automated method (called DynaHug[1]) that learns the behavior of benign models using a combination of dynamic analysis and ML. To the best of our knowledge, DynaHug is the first technique that automatically learns to detect malicious models via dynamic behavioral analysis. Figure 1 and Algorithm 1 present the design and algorithm of our DynaHug approach. DynaHug learns the behaviors of benign models by first clustering models by tasks (e.g., `text-classification`). Secondly, it conducts dynamic analysis of the model in a sandbox and collects system call traces. In this step, PTMs are executed by loading the model and deserializing it. Next, it trains a one-class SVM that learns the behavior of benign models. At inference, when given the system call traces of a PTM under

1. DynaHug means "**Dyna**mic **Hug**ging Face PTM Detector"

TABLE 1: Details of state-of-the-art detectors versus our approach (DYNAHUG) showing whether the detector "fully" (●), "partially" (◐), or "does not" (○) employ the specified analysis technique.)

Detection Tools	Rule Based	Import Scanning	Blacklist	Whitelist	Heuristic	Dataflow Analysis	Vulnerability Detection	Restricted Loader	Machine Learning	Dynamic Analysis	Open Source
PickleScan [20]	●	●	●	◐	○	○	○	○	○	○	●
ModelScan [21]	●	●	●	○	○	○	○	○	○	○	●
Fickling [24]	●	●	●	◐	○	●	○	○	○	○	●
MALHUG [8]	●	●	●	○	●	○	○	○	○	○	○
PickleBall [25]	◐	●	○	●	●	○	○	●	○	○	●
ModelTracer [23]	●	○	●	○	○	○	○	○	○	●	●
Weights-Only [26]	●	●	○	●	○	○	○	●	○	○	●
JFrog (HF) [19]	●	●	●	○	○	○	○	○	○	○	○
Guardian (HF) [17]	●	●	●	○	○	○	◐	○	○	○	○
ClamAV (HF) [18]	○	○	●	○	○	○	●	○	○	○	●
HF_PickleScan [18]	●	●	●	◐	○	○	○	○	○	○	○
DYNAHUG	○	○	○	○	○	○	○	○	●	●	●

test (PUT), DYNAHUG automatically detects whether it is malicious or benign.

This work makes the following contributions:

1) **DYNAHUG**: We propose an automated method (called DYNAHUG) that learns to detect malicious PTMs by learning the behavior of task-specific clusters of benign models via dynamic program analysis. DYNAHUG provides an automated learning method for detecting malicious PTMs in Model Hubs such as Hugging Face.
2) **Evaluation**: We evaluate DYNAHUG using over 18,000 PTMs. Our experiments demonstrate that DYNAHUG is effective in detecting malicious models with an F1-score of up to 0.9963 across all datasets.
3) **State-of-the-art Comparison:** We compare DYNAHUG to five (5) PTM detectors – PickleScan, ModelScan, Fickling, ModelTracer and an LLM-based detector (Llama-3.1-8B-Instruct-tuned). Results show that DYNAHUG is up to 44% more effective than the baselines.
4) **Ablation and Sensitivity Studies:** We report ablation studies examining whether our design decisions (e.g., dynamic analysis, etc.) contribute positively to DYNAHUG's effectiveness and outperform alternative design choices. Additionally, we conduct probing and sensitivity studies examining the impact of task clustering, and training sizes on DYNAHUG's performance.

2. Overview

2.1. Problem Definition

In this work, we pose the following scientific question: *Given a PTM, how can we automatically detect that it is malicious (or benign)?* Addressing this question is important to ensure the security of trusted execution environments and Model Hubs. Specifically, we aim to develop an automated method that ensures that malicious PTMs are identified during model execution or when uploaded on Model Hubs. We consider a PTM to be *malicious* if it exhibits insecure or unsafe behaviors that are beyond the intended behaviors of typical PTMs. For instance, PTMs that copy user data, sends user data to an unknown IP address or perform remote code execution are considered to be malicious [32, 33]. Thus, a malicious model detector is effective, if it accurately detects genuinely malicious models and does not misclassify benign models. Otherwise, we consider a detector to be ineffective.

2.2. Key Insight

This work proposes DYNAHUG, an automated technique for malicious PTM detection. The main idea of DYNAHUG is to employ dynamic program analysis and ML to learn the behavior of benign PTMs. The *key insight* of our approach is that *malicious PTMs often exhibit behaviors that are unique, rare or different from the behaviors of benign models.* In particular, we posit that for a specific ML task, (a) benign PTMs exhibit common behaviors, e.g., they often use similar system calls (with similar frequency) and (b) malicious PTMs often exhibit unique, outlier behaviors that are rarely exhibited by benign PTMs. Hence, we hypothesize that automatically learning the behaviors of benign PTMs is applicable for detecting malicious PTMs. Malicious PTMs are known to perform certain operations (e.g., remote code execution) that are never or rarely performed by benign PTMs [8]. This is evident from the fact that malicious payloads require certain system calls for attack orchestration. For instance, malicious PTMs that perform remote code execution often employ system calls (such as `execve`) which are rarely used by benign models since PTMs do not typically execute arbitrary code or require shell access [23]. This insight is evident in the literature; previous works have shown that certain behaviors (e.g., imports or system calls) are security sensitive, unique to malicious models or identifiable as safe, unsafe or sensitive [8, 23].

2.3. Motivating Example

Table 2 shows examples of real-world benign and malicious PTMs. It highlights the disassembled codes and dynamic traces of each model (third and fourth row). Malicious payloads/traces are highlighted in red text (third to fourth column - "MALHUG" and "PyPI"). Benign code snippets and traces that are often mis-classified as malicious by most detection tools are highlighted in orange text ("Benign"

TABLE 2: Motivating Example showing benign and malicious PTMs and the performance of DynaHug and baselines. Disassembled code/trace snippets in orange are benign (e.g., unseen imports or system calls missing in the whitelist of Fickling and hence mis-classified). Code/trace snippets in red shows malicious payloads ✓ mean the model was detected by a tool as an anomaly, ✗ mean the model was classified as benign and ✓indicates a false positive.

	Benign	**MalHug**	**PyPI**
Model Name	llm-stacking/G_learn_depth [27]	Narsil/totallysafe [28]	Zolllll/dont_download_this [29]
Description of Model	Benign model importing a Tensor class from torch, not part of the standard library [30]	Model found in MalHug [8] containing blacklisted library compile [31]	Model injected with a payload using library execute from PyPI [29]

Static Opcodes	`110: \x93 STACK_GLOBAL`
	`111: \x94 MEMOIZE (as 7)`
	`112: \x8c SHORT_BINUNICODE 'torch'`
	`119: \x94 MEMOIZE (as 8)`
	`120: \x8c SHORT_BINUNICODE 'Tensor'`
	`128: \x94 MEMOIZE (as 9)`
	`129: \x93 STACK_GLOBAL`
	`130: \x94 MEMOIZE (as 10)`
		`0: c GLOBAL 'os system'`
		`11: ( MARK`
		`12: S STRING 'sleep 5 && find ~ -iname "*" -exec echo Removing {} \\; -exec sleep 1 \\; &'`
		`90: t TUPLE (MARK at 11)`
		`91: R REDUCE`
			`0: \x80 PROTO 2`
			`2: c GLOBAL 'execute both'`
			`16: q BINPUT 0`
			`18: X BINUNICODE 'nc -e /bin/sh 127.0.0.1 4444'`
			`51: q BINPUT 1`
			`53: \x85 TUPLE1`
			`54: q BINPUT 2`
			`56: R REDUCE`
			`57: q BINPUT 3`

Dynamic Traces	`104089 newfstatat(AT_FDCWD, "/usr/src/app/.venv/lib/python3.10/site-packages/torch/_tensor.py", {st_mode=S_IFREG|0644, st_size=74282, ...}, 0)= 0`
	`104089 newfstatat(AT_FDCWD, "/usr/src/app/.venv/lib/python3.10/site-packages/torch/_tensor.py", {st_mode=S_IFREG|0644, st_size=74282, ...}, 0)= 0`
	`...`
	`104089 socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC, IPPROTO_IP) = 3`
	`104089 bind(3, {sa_family=AF_INET6, sin6_port=htons(0), sin6_flowinfo=htonl(0), inet_pton(AF_INET6, "::1", &sin6_addr), sin6_scope_id=0}, 28) = 0`
		`322298 newfstatat(AT_FDCWD, "/usr/sbin/find", 0x7ffdf9cfb510, 0) = -1 ENOENT (No such file or directory)`
		`322298 newfstatat(AT_FDCWD, "/usr/bin/find", {st_mode=S_IFREG|0755, st_size=224848, ...}, 0) = 0`
		`322298 execve("/usr/bin/find", ["find", "/root", "-iname", "*", "-exec", "echo", "Removing", "", ";", "-exec", "sleep", "1", ";"], 0x591a72855198 /* 48 vars */) = 0`
		`322298 brk(NULL) = 0x5c7c67f91000`
		`322298 access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)`
			`...`
			`410421 close(7) = 0`
			`410421 close(9) = 0`
			`410421 getdents64(5, 0x7ffcf8560c80 /* 0 entries */, 280) = 0`
			`410421 close(5) = 0`
			`410421 execve("/bin/sh", ["/bin/sh", "-c", "nc -e /bin/sh 127.0.0.1 4444"], 0x611b65855370 /* 46 vars */ <unfinished ...>`
			`410416 <... vfork resumed>) = 410421`
			`410416 rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0`
			`410421 <... execve resumed>) = 0`
			`410416 close(11) = 0`
			`...`

Fickling[24]	✓ (Tagged as unsafe)	✓	✓ (Tagged as unsafe)
PickleScan [20]	✗	✓	✗
ModelScan [21]	✗	✓	✗
HF_JFrog [19]	✗	✓	✗
HF_Guardian [17]	✗	✓	✗
HF_ClamAV [18]	✗	✗	✗
HF_PickleScan [18]	✓(marked as needing attention)	✓	✓(marked as needing attention)
ModelTracer [23]	✓	✓	✓
DynaHug (our work)	✗	✓	✓

second column). The detection performance of DynaHug and state-of-the-art detectors are reported (in the last nine rows) showing true positive detection (✓), no detection (✗) and false positive detection (✓).

The two malicious models (last two columns) allow an attacker to conduct dynamic execution of arbitrary Python code during model loading or deserialization. As shown in Table 2 ("MALHUG" third column), most tools detect the malicious payloads involving builtin Python library imports (e.g., `os`). In particular, such imports are often used to orchestrate dynamic code execution in Python. Thus, they are typically added to the blacklist rules of most detectors.

However, most tools are unable to detect uncommon or rare PyPI modules that allow for dynamic code execution ("PyPI" last column). State-of-the-art detectors miss such modules due to the non-exhaustive nature of blacklists. Indeed, there are numerous PyPI libraries that allow for dynamic code execution which makes it impossible to construct a complete blacklist.[2] Additionally, we show that some detectors (e.g., Fickling) flag almost all models as malicious. This is due to its non-exhaustive set of safe and unsafe opcodes and library imports. Overall, state-of-the-art detectors suffer from under- or over- approximation due to static, incomplete sets of blacklisting or whitelisting rules.

Unlike most detectors, DynaHug detects the two malicious models in Table 2 and it correctly classifies the benign model. This is due to its use of *task-specific clustering*, *dynamic analysis* and *ML*, which allows it to accurately learn the behavior of benign models and flag the behaviors of malicious models: In DynaHug, dynamic analysis allows to collect behaviors of benign models, and ML allows to generalize learned behaviors beyond a static rule set. DynaHug is able to detect malicious PTMs using rare library imports (Table 2 column 1), since such imports use specific system calls which are rarely used by benign models. This demonstrates the importance of dynamic analysis. Our task-

2. The first 20 pages of PyPI when searching "execute" [34] contains over 20 modules that can perform reverse shell execution, many of which are not blacklisted by the state-of-the-art detectors (as at 10 Oct 2025).

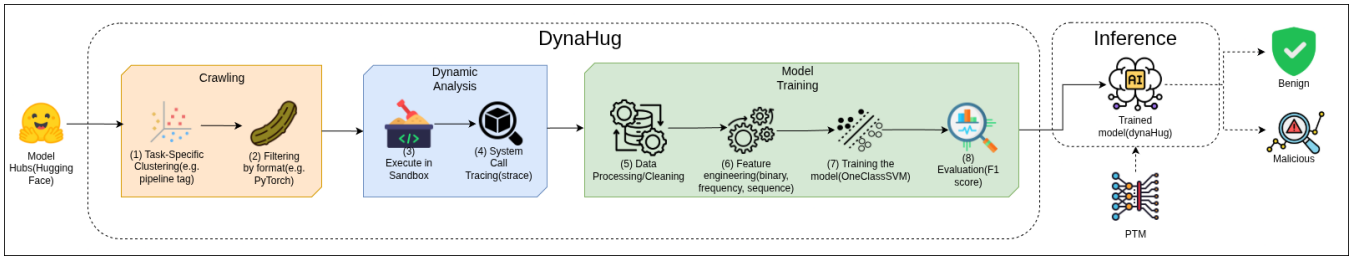


Figure 1: Workflow diagram of DYNAHUG

specific clustering allows to narrow down the set of behaviors that are unique to a specific category of PTMs. This is inspired by previous works in anomaly detection which have demonstrated that specific categories of applications exhibit similar program behaviors [35, 36, 37].

2.4. Novelty vs. State-of-the-art

Table 1 illustrates the novelty of our approach (DYNAHUG) with respect to state-of-the-art techniques. Detectors can be divided into two classes, namely static detectors and dynamic detectors. On the one hand, static detectors employ static analysis (e.g., disassembling, dataflow analysis or tainting) alongside blacklisting or whitelisting of specific imports, system calls or opcodes. For instance, PickleScan [20] employs a combination of static analysis, blacklisting and whitelisting of safe and unsafe imports and opcodes for malicious PTM detection (*see* Table 1). On the other hand, there are few dynamic detectors. Notably, ModelTracer [23] combines dynamic analysis and a blacklist of system calls to detect malicious PTM. However, ModelTracer's blacklisting limits it from capturing rare or previously unseen imports or system calls and does not allow for subtlety in blacklisted calls. As exemplified in Table 2 (column two), ModelTracer detects the benign model as malicious because of the presence of a `socket` call, which is part of the blacklist. Meanwhile, DYNAHUG is able to correctly classify the model as benign since it does not rely on a rulelist. The system level granularity of DYNAHUG's traces further contributes to its performance since the hundreds of imports/opcodes that are manually blacklisted or whitelisted by existing approaches translate to a limited set of system calls during DYNAHUG's behavioral analysis.

To the best of our knowledge, DYNAHUG is the first approach that employs ML and dynamic analysis for malicious PTM detection. DYNAHUG is unique with its combination of task-specific clustering, dynamic analysis and machine learning. As shown in Table 1, DYNAHUG is the *only* tool that does not rely on blacklisting or whitelisting rules to detect malicious PTMs and the *only* detector that deploys ML for malicious PTM detection. Besides, DYNAHUG is only one of two tools that employs dynamic analysis.

2.5. Threat Model

Attack assumptions: In this work, we assume the attacker modifies an existing model or acts as a third-party model provider supplying a PTM containing arbitrary malicious payloads, e.g., reverse shell. The malicious model is distributed through Model Hubs, provided online or directly sent to users. The attacker provides the malicious model as well as instructions to load/execute the model. This attack setting is practical and fits how models are provided on Model Hubs (Hugging Face [1], GitHub [3] and Kaggle [2]). **Defense assumptions:** We assume the defender loads the model following the instructions provided by the model provider. We do not require access to the source code of the model or need security expertise (e.g., to modify how the model is loaded), except the instructions provided by the provider (e.g., in the model card or README). We do not assume access to module libraries or need to modify the default model loading instruction. We assume the model under test is executable in a sandbox and the attacker does not employ anti-debugging techniques. These assumptions are practical, relying on the trust of the victim in the Model Hubs and third-party vendors and does not require additional security or ML expertise or external tool except the model and the model loading instructions. This setting is realistic and similar to the manner end-users obtain PTMs from Model Hubs – Hugging Face [1], GitHub [3] and Kaggle [2].

3. Methodology

3.1. DYNAHUG Overview

Algorithm 1 describes the DYNAHUG algorithm. Figure 1 also illustrates the high-level workflow of DYNAHUG with each component color mapped to the DYNAHUG algorithm. The main three (3) steps of DYNAHUG are color mapped for both the algorithm (Algorithm 1) and the workflow diagram (Figure 1): *Crawling* steps are highlighted in orange, *Dynamic Analysis* steps in blue and *Model Training* steps in green.

The goal of DYNAHUG is to learn the benign behavior of PTMs for a specific ML task (e.g., text generation). It achieves this via dynamic analysis and machine learning. Given a PTM under test (PUT) and the task tag of the PTM (e.g., text generation), DYNAHUG learns the benign behavior of benign models in the task tag and identifies the PUT to be either *malicious* or *benign*.

To achieve this, DYNAHUG first fetches benign PTMs for the task at hand from a Model Hub (e.g., Hugging Face) by filtering for the task tag and the model format (e.g, Pickle) (Algorithm 1:line 2). Once a large number (2K) of benign PTMs are retrieved, DYNAHUG loads (deserialises) each model in a sandbox and collects its system call

traces using runtime observability tools like `strace` (Algorithm 1:line 5,6). It then processes the collected traces into meaningful features, transforming them into a structured and interpretable format (Algorithm 1:line 11). Once the traces are processed, it engineers new features from the existing data, i.e., binary and frequency features (Algorithm 1:line 12). Next, DYNAHUG learns an ML model detector by first tuning the model to maximize performance by passing this processed data for training alongside the hyperparameters and ML architectures (Algorithm 1:line 15,16,19). Using the F1-score as the main evaluation metric, the best performing model is then chosen as DYNAHUG. Finally, at inference, DYNAHUG detects whether the *PUT* is benign or malicious. It first collects and processes the trace data of the PUT (similarly to the *Model training* step), then passes the processed data for a detection output of *malicious* or *benign*.

Algorithm 1 DYNAHUG

Input: TAG, modelToAnalyze
Output: Malicious or Benign classification
1: **// Phase 1: Crawling**
2: $M_{\text{tag}} \leftarrow$ fetchModelsFromHub(type = "PyTorch", TAG)
// Phase 2: Dynamic Analysis
3: traces $\leftarrow \emptyset$
4: **for** $M_i \in M_{\text{tag}}$ **do** ▷ Model deserialization
5: executeModelInSandbox(M_i)
6: traces $\leftarrow$ traces $\cup$ monitorRuntimeBehaviour(M_i)
7: **end for**
8: **// Phase 3: Model Training**
9: $X \leftarrow \emptyset$ ▷ Training dataset
10: **for** trace$_i \in$ traces **do**
11: $d_i \leftarrow$ dataProcessing(trace$_i$)
12: $X \leftarrow X \cup$ featureEngineering(d_i)
13: **end for**
14: $\mathcal{M} \leftarrow \emptyset$ ▷ List of classifiers to evaluate
15: modelArch $\leftarrow$ OneClassSVM
16: hyperparams $\leftarrow \{kernel : [\texttt{RBF}, \texttt{linear}, ...], ...\}$
17: **for** $h \in$ hyperparams **do** ▷ Grid Search CV
18: $\mathcal{M} \leftarrow \mathcal{M} \cup$ trainModel(X, modelArch, h)
19: **end for**
20: $(m_{\text{best}}, h_{\text{best}}) \leftarrow$ evaluation($\mathcal{M}$) ▷ DYNAHUG

3.2. Detailed Methodology

3.2.1. Crawling and Clustering. The goal of this phase is to collect relevant models from a Model Hub. We make use of the proprietary Model Hub API (e.g., `huggingface_hub [38]`) to filter the repositories based on the tag and the type of the model artifact as shown in Algorithm 1. Prior works, such as Gorla et al. [35], demonstrate that Android applications that are similar, in terms of their descriptions, should also behave similarly. Analogously, our underlying assumption is that models which are assigned the same tags should behave similarly. Task tags in Model Hubs identify the task a particular model is designed for. For instance, the `text-generation` pipeline tag in HF identifies models that are designed for text generation tasks. This helps narrow down the expected behaviour of a benign model from that particular tag.

In our analysis of HF, we rank the models according to the likes, and filter models based on pipeline tags with the help of the `huggingface_hub [38]` library in Python. The intuition is that popular models, models which have a higher amount of "likes", are less likely to be malicious. We focus our efforts on PyTorch models since 95% of all malicious models in HF are known to be PyTorch models [39]. Hence, when retrieving the models, we make sure that a `pytorch_model.bin` is present. We look for this file in particular since it is common convention within Hugging Face (HF) to store the model weights for inference inside `pytorch_model.bin`. This is to ensure that the `transformers` library from HF is able to find the weights file when trying to load a model for inference [40]. Next, the fetched models are stored in a Google Cloud Storage (GCS) Bucket [41] and sent to a sandbox for dynamic analysis.

3.2.2. Dynamic Analysis. DYNAHUG simulates an inference workflow of a user trying to run a model downloaded from a Model Hub. We build a `Docker` [42] container to load/deserialize the PyTorch models, emulating a reproducible environment similar to that of a user. `Docker` containers provide OS-level virtualization to isolate any processes running within it from the host machine, preventing any damage from potentially malicious models, e.g., Remote Code Execution (RCE) on the host machine [43]. Moreover, since the output logs from strace are non-deterministic and varies with the environment on which the PyTorch model is deserialized and the hardware resources available at that point in execution, it is imperative to have control on the environment, which `Docker` provides [44]. DYNAHUG provides the flexibility of choosing the runtime observability tools (e.g., `strace` [45], `tcpdump` [46], `eBPF` [47], etc.) to the user depending on the runtime aspects they wish to monitor. Once the runtime observability tool records the deserialisation process of the PyTorch model, the logs are saved for further processing in the upcoming phases.

In our analysis, we utilize `strace`, a diagnostic userspace utility used to monitor interactions between processes and the Linux kernel which includes system calls, signal deliveries and change of process state [45]. `strace` monitors system calls related to File System Management, Networking, Process Management, Memory Management, etc. which covers almost all key operations to give a broad overview of system activity. Previous works [23] also utilize `strace` for dynamic analysis and detection of malicious PTMs. `strace` outputs a detailed raw log file which contains the different system calls and their arguments. For instance, Example 1 shows the system call, `execve`, running the unix tool, `cat`, to display AWS secrets on the terminal output in a folder commonly known to store them.

Example 1: Strace log snippet from deserializing unsafe_model.pkl (ankushvangari/unsafe_model [32])

```
22928 execve("/usr/bin/cat", ["cat",
    "/home/sandbox/.aws/secrets"], 0x5d0f8b34df28 /* 55
    vars */ <unfinished ...>
22927 <... vfork resumed>)              = 22928
22927 rt_sigprocmask(SIG_SETMASK, [], ~[KILL STOP RTMIN
    RT_1], 8) = 0
22928 <... execve resumed>)             = 0
```

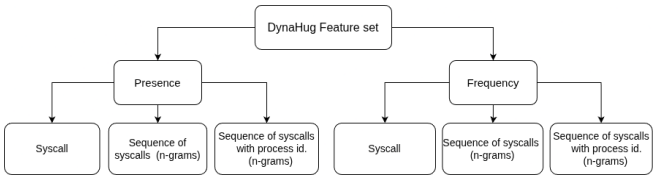


Figure 2: Experimental feature set of DYNAHUG

3.2.3. Model Training. To train our classifier effectively, it is important that the input data be represented in a manner that the model can interpret efficiently.

Data Processing: DYNAHUG extracts meaningful information such as system call frequency and stores it in structured formats such as CSV and Apache Parquet.

Feature Engineering: DYNAHUG prepares a presence and frequency feature set for system calls as shown in Figure 2 to feed into the classifier. Presence features indicate the presence of a certain feature with a binary value of either 0 or 1. Frequency features indicate the frequency of occurrence of a certain feature. These features would provide a detailed insight on the statistical distribution of system calls during the deserialization process. To avoid bias towards certain system calls due to their sheer magnitude, DYNAHUG normalizes frequency features, so that all system calls are on the same scale.

In our analysis, we utilize the `-c` flag in `strace` to get the summarized information of system calls (Section 3.2.2) during deserialization. The frequency of system calls is parsed from this summarized data and stored in Apache Parquet, since it is faster to load than CSV [48]. DYNAHUG fetches the presence and frequency features for plain system calls and applies normalization with the help of `StandardScaler(nomean=True)` class in `scikit-learn` [49]. Next, the established feature set is passed to the training component. We have also evaluated other features, e.g., n-grams and Process IDs (*see* section 4).

Training: DYNAHUG performs hyperparameter tuning while training a one-class SVM. The models trained from each parameter setting are evaluated using F1-score. The best performing model is chosen as DYNAHUG and used during inference. At inference, DYNAHUG follows a similar approach to training to retrieve and process the data from the model analysis (Section 3.1). The processed data is passed into DYNAHUG for detection – ”Benign” or ”Malicious”.

4. Experimental Setup

4.1. Research Questions

- **RQ1 Effectiveness:** How effective is DYNAHUG in detecting malicious PTMs?
- **RQ2 State-of-the-art Comparison:** How does DYNAHUG compare to the state-of-the-art detectors?
- **RQ3 Ablation study:** What are the contributions of our design decision (e.g., static vs. dynamic features) to the performance of DYNAHUG?
- **RQ4: Probing and Sensitivity Study:** What is the impact of clustering on DYNAHUG's performance? How sensitive is DYNAHUG to different training dataset sizes?

4.2. Datasets

In our experiments, we employ Hugging Face (HF) as our Model Hub, since it is the most popular model hub with the largest set of models. Table 3 provides details of the top six model hubs. We collect our dataset of benign models from the Top-K (default 2000) models on HF. Our experiments involve two task categories from HF, namely *text generation* and *text classification*.

Table 4 describes our collected dataset for each cluster. For the text-generation cluster, we first filter out models that are flagged by HF as malicious and that are not part of known malicious PTMs [8, 25]. In the top 3,000 models, we found and filtered out two (2) PTMs. Specifically, we download the top 3000 models ranked by most likes, from which we filtered out two (2) models that are flagged by HF and MALHUG [8] as malicious. We download another 1000 benign models so that we can evaluate DYNAHUG on them, and also inject them with malicious payloads to make our injected set of 2000 models. To leave an appropriate gap between the training set and the injection set to avoid contamination, we collect our next set of 1000 models from the 3600 to 4600 top liked models. Furthermore, we download the next 1025 models that are not flagged by HF as an evaluation set to test the model against, and balance our dataset with regards to our malicious set. The above setting describes all text-generation experiments (**RQ1** to **RQ4**). Then, we repeat the steps described above for our text-classification (**RQ1**) and non-clustered experiments (**RQ4**).

4.3. Injected malicious models

To ensure detectors generalise to unseen models and address class imbalance, we augment our evaluation dataset by injecting malicious payloads into benign models using models from HF and malicious payloads from MALHUG and libraries from PyPI that allow for code execution.

MALHUG injected models: Using Fickling [24], we automatically inject 64 malicious payloads from MALHUG into 1000 benign models from HF. Example 2 shows an example

TABLE 3: Number of PTMs on Top six (6) Model Hubs. N/A indicates task not categorised on the platform.

Hub	#Models	Formats	Text Generation	Image Classification	Text Classification	Translation	Image Segmentation	Feature Extraction
Hugging Face [1]	1881296	Pytorch, Tflow, gguf	296411	19009	98414	6065	1801	13864
Github [3]	150685	Pytorch, Tflow, gguf	1732	9019	4427	7080	2494	1957
Spark NLP [5]	135372	Custom	25181	8643	35248	30903	N/A	N/A
OpenCSG [4]	111189	Pytorch, Tflow, gguf	4224	840	278	491	79	484
ModelScope [6]	80574	Pytorch, Tflow, gguf	20307	2148	1172	260	240	740
Kaggle [2]	3140	Pytorch, Tflow, gguf	800	907	197	N/A	212	N/A

TABLE 4: Details of training and evaluation dataset including Benign and Malicious across all clusters.

Datasets	Range	Benign			Malicious			Total
	(Sorted By Most Likes)	Text Gen	Text Class	Non Cluster	Text Gen	Text Class	Non Cluster	
Hugging Face	1-3000	3000	3000	3000	3	0	0	9003
Injected	3600-4600	1000	1000	0	1000	1000	1000	5000
Injected PyPI	3600-4600	0	0	0	1000	1000	0	2000
Evaluation	4600-5625	1025	1025	0	0	0	0	2050
MALHUG	-	0	0	0	20	2	84	106
Pickleball	-	0	0	0	2	2	0	4
Total	5625	5025	5025	3000	2025	2004	1084	18163

of a MALHUG injected payload. We ensure a model is valid by checking that it is successfully loaded by PyTorch with `torch.load()`. Injection process times out and proceeds to the next payload when there is a blocking operation from a MALHUG payload, e.g., it requires user input (`input()`).

PyPI injected models: We automatically inject 20 malicious payloads using PyPI libraries [50] into 1000 benign models from HF. The goal of the PyPI injected model is to provide more realistic malicious paayloads (e.g., used for information reconnaissance, or reverse shells), since most MALHUG payloads are mostly Proof-Of-Concepts (PoCs). Example 3 shows an example of our PyPI injected payload. These libraries allows the user to either execute system commands (similar to `os.system()`), or execute Python code (similar to `exec()`). Using these 20 malicious pickles, we directly inject their Pickle bytecode into the 1000 benign models we collected, after modifying the required stack addresses such that there is no memory conflict between the original model and the injected payload.

For each injected payload, we setup a test oracle to confirm that the resulting models are valid, and injection was successful. To account for the blocking operations of the reverse shells, we start a netcat [51] server that sends an `exit` instruction to terminate the reverse shell.

4.4. Dataset collection and crawling setup

DYNAHUG utilizes functions from the `huggingface_hub [38]` library to fetch a list of repository names from HF to crawl. We pass *likes* and *pipeline tag* as arguments to this function to obtain a list of repositories ranked in descending order of *likes* and clustered based on the *pipeline tag* in HF. By iterating through each repository in the list, we obtain metadata information such as the type of files being stored. We look through the list of files at the top level of the repository and check for the presence of `pytorch_model.bin`. For each PyTorch model file, we check its HF security status and skip it if the model file was identified by HF as "unsafe". We retrieve the size of the file and estimate the memory

Example 2: Example MALHUG payload [52]

```
2: c    GLOBAL     '__builtin__ eval'
20: q    BINPUT     0
22: X    BINUNICODE
 'exec(\'\'\'\nprint("Hello, I am Gorden
 Ramsey!")\n\'\'\') or dict()'
```

Example 3: Example PyPI payload

```
2: c    GLOBAL     'raft run'
12: q    BINPUT     0
14: X    BINUNICODE "zsh -c 'zmodload zsh/net/tcp &&
 ztcp 127.0.0.1 4444 && zsh >\&$REPLY 2>&$REPLY
 0>\&$REPLY'"
```

usage when it is deserialized during dynamic analysis to avoid Out Of Memory (OOM) issues. If the memory safety check succeeds, the `pytorch_model.bin` file is downloaded from the repository and we check that it contains a Pickle file.[3] If all the aforementioned conditions are met, we store any relevant metadata about the repository in a CSV file, i.e., number of likes, downloads, date of last commit and HF security tool [17, 18, 19] detection.

4.5. Training Process

Architecture Selection: In our analysis, the one-class SVM (OCSVM) was selected since it is a popular choice in unsupervised learning based anomaly detection, alongside Stochastic Gradient Descent one-class SVM (SGDSVM) and Isolation Forest (IF). These architectures require only one-class of training data, which becomes essential in the case of a dataset imbalance. ML algorithms supporting one-class training are necessary in the PTM setting since there are millions of benign models but few malicious models.

Feature Engineering: DYNAHUG prepares the feature set needed to be fed into the classifier for training. Figure 2 shows the variety of features that were used in the experiments. When constructing features related to individual system calls, a platform-specific exhaustive list of system calls was used, even if some of the system calls were not invoked in any of the runs in dynamic analysis. This was adopted to prevent Out Of Vocabulary issues that could arise from unseen system calls in the test set. The exhaustive list was obtained from the Linux kernel source code in the docker container [53], corresponding to the x86-64 architecture.

DYNAHUG has three (3) different features sets for training. First, *presence and frequency* features capture

3. This step is required since `.bin` files do not guarantee the presence of `.pickle` or `.pkl` files.

the presence and frequency of individual system calls (e.g.,"execve", "read", "write"). *Sequence* based features represent the presence and frequency of system call sequences (n-grams) (e.g., "execve:read", "read:write"). *Sequence* features are represented with 2-grams, as this configuration offers the best trade off between model complexity and performance. Although higher order n-grams (i.e., 3-gram, 4-gram) provide a better spacial understanding of system calls to the model, preliminary analysis showed that choosing these n-gram configurations would lead to poorer performance. Finally, *Process Sequence* based features incorporated presence and frequency of process ID (PID) system call sequences (e.g., "P1:close_P5:rt_sigprocmask", "P4:rseq_P1:read" where P1, P4 and P5 are generalized process IDs). The raw PID value is not used to represent the process since each process has a unique PID in every run which would introduce noise into our training dataset. Hence, a generalized form of the PID is used where the parent process is P1 and any subsequent processes would be numbered accordingly. Similar to sequence features, process sequence features are also 2-grams.

Dataset Split: The dataset containing the features is split into train, validation and test sets in the ratio 80:10:10.

Hyperparameter Tuning Strategy: DYNAHUG performs Grid Search on a set of hyperparameters which are tailored to a specific classifier architecture. We employ a 5-fold cross validation procedure to ensure that we get the average F1-score for the best model evaluation.

Hyperparameter Choices: For OCSVM, we tune parameters *nu* (`0.01` to `0.5`), *kernel* (`linear`, `RBF` and `sigmoid`), and *gamma* (`0.01`, `0.1`, `1` and `auto`). For the SGDSVM, we additionally tune batch_size (`32`, `64`, `128` and `1000`). Finally, for IF, we tune *n_estimators* (`50` to `150`), *contamination_rate* (`0.001`, `0.01`, `0.1` and `auto`) and *max_samples* (`128`, `256`, `512` and `1000`).

Data Preprocessing: Once the hyperparameter combination is selected, we utilize the `DictVectorizer` from `scikit-learn` [49] to transform the feature-value mapping to a matrix which is comprehensible to the classifier. The values within this matrix are normalized by applying the `StandardScaler(nomean=True)` class. The centralization of the values by the mean is disabled to prevent the data from losing its sparse nature and optimizing memory usage by avoiding the sparse matrix from becoming a dense matrix. This normalized data is fitted to the classifier of choice and this process is repeated until a classifier with the highest possible average F1-score on the validation set is selected as the inference-ready model. The inference-ready model is also evaluated on the test set to gain deeper insights of the model's performance on unseen data.

4.6. Metrics and Measures

DYNAHUG uses accuracy, precision, recall and F1-score to evaluate trained models. The F1-score was utilized as a judging criteria for selecting the best model during hyperparameter tuning. The F1-score is the harmonic mean between precision and recall. This means that an increase in False Positives (affects Precision) or False Negatives (affects Recall) would punish the F1-score, promoting a balance between False Positives and False Negatives. This judging criteria encourages the hyperparameter tuning to choose a model which has high precision and recall scores.

4.7. Research Protocol

Firstly, we evaluate DYNAHUG using two different tasks (**RQ1**) by training our classifer on the `text-classification` and `text-generation` cluster. The goal is to measure the effectiveness of our approach on different clusters. Besides, we compare DYNAHUG to the current state-of-the-art (**RQ2**), we evaluate existing open-source detectors encompassing both static and dynamic analysis. We also evaluate whether LLMs can serve as an effective malicious PTM detector by replacing our classifier with an LLM and using the traces collected from dynamic analysis as an input. Next, we conducted an ablation study (**RQ3**) examining our design choices. We examined the contribution of (a) dynamic analysis versus alternatives (i.e., static, dynamic or hybrid), (b) ML architecture (i.e., OCSVM, SGDSVM, and IF) and (c) feature set provided to the classifier for training and inference (i.e., *presence, frequency, sequence, process sequence*). Additionally, we investigate the performance of DYNAHUG with and without task clustering (**RQ4**). Finally, we performed sensitivity analysis (**RQ4**) of the classifier against varying training dataset size, i.e., top 1000, 2000 and 3000 repositories from HF.

4.8. Baseline Selection

Security Scanners: We chose PickleScan [20] and Modelscan [21] as they are the closest open-source alternatives to the currently used scanners on Hugging Face [17, 18]. PickleScan is the base version of Hugging Face's HF PickleScan, while Modelscan is developed by the same company, ProtectAI, which also developed Hugging Face's Guardian. We also use Fickling [24], developed by Trail of Bits, as it provides novel detection methods. To the best of our knowledge, ModelTracer [23] is the only open-source dynamic scanner for malicious PTM detection.

LLM baseline: We employ Llama-3.1-8B-Instruct-tuned (Llama-3.1) for the LLM baseline since the Meta-Llama series of LLMs is widely recognized as one of the state-of-the-art open source models and it has demonstrated strong capabilities in security-related applications [54, 55, 56]. It is free to use, computationally efficient and instruction-tuned to produce standardized outputs.

Excluded Scanners: In our baseline comparison (**RQ2**), we exclude the following types of scanners because they are not applicable to our threat model, impractical in practice, or impossible to execute on our dataset, namely the following: *a.) Hugging Face Scanners:* We note that Hugging Face (HF) runs closed-source versions of security scanners on

uploaded models. However, we do not evaluate against the HF scanners as we already use them to filter for models that are benign and thus would be inhenerently biased. We also cannot feasibly evaluate the HF Scanners against our injected malicious set as the only way to run them would be to upload them to HF –calling for the need of upload of 6000 injected malicious models– which comes with a risk of violating the platform's policies.

b.) Restricted Loading Environments (RLEs): We determined that RLEs were not applicable to our threat model, as they require a security-aware user. Specifically, we note that Weights-Only Unpickler [26], developed by PyTorch, can be bypassed by merely instructing the user to set `weights_only=False` while loading the model. Furthermore, we also understand that Pickleball [25] overrides the default Pickle and PyTorch `load()` functions in the Docker container provided by the authors. Besides, it relies on the users' security expertise to generate the required policies for undocumented libraries.

4.9. Baseline setting

We compare DYNAHUG to the open-source state-of-the-art (SOTA) malicious PTM detectors. In particular, three static detectors (`PickleScan` [20], `Modelscan` [21], `Fickling` [24]) and one dynamic detector (`ModelTracer` [23]). We also compare DYNAHUG to an open-source SOTA LLM, namely Llama-3.1 [57].

Baseline Scanner setup: We execute PickleScan and ModelScan via a Python script that calls a shell command to run the tool, given the path to the PUT. To run Fickling, we employ `analyze_pickle_safety`, to obtain various security threat levels (such as `LIKELY_UNSAFE`, `OVERTLY_MALICIOUS`). To run ModelTracer, we execute the code provided by its Github repository [58].

LLM baseline setup: For the LLM baseline evaluation, Llama-3.1 was provided raw traces obtained from dynamic analysis as its input. This design choice was made since LLMs are known to have a strong capability to develop a rich semantic understanding of text, enabling it to have a deeper insight into the relationship between the different system calls which occur within an `strace` [45] log file. Due to the small context window of Llama-3.1-8B-Instruct-tuned [57] (i.e., maximum 128k tokens [59]) and the limited CUDA memory of the NVIDIA A100 chip (i.e., 64GB CUDA memory [60]), passing in the `strace` logs in its entirety would leave the program susceptible to either CUDA OOM error or the LLM losing out on vital information for accurate detections. To mitigate this, we filter out which system calls are needed to be passed for the LLM to have an understanding of the overall functioning of the deserialized PyTorch file. We curated a list of 35 system calls borrowed from Brown et al. [61] alongwith an additional 23 system calls to filter the `strace` logs. The LLM was instructed with Few-Shot prompting [62] to use this filtered `strace` log to look for any red flags during pickle deserialization. Example 4 portrays the system prompt and a sample LLM response. To mitigate randomness, the `temperature` of the LLM was set to zero (0).

Example 4: System prompt and LLM response

```
Prompt:
You are analyzing strace logs to detect malicious
    activity in PyTorch files.

Look for RED FLAGS like these:
- Process creation (execve,fork,clone,...)
- Network connections (socket, connect, bind, ...)
- File writes outside normal directories
- Access to sensitive files (/etc/passwd, SSH keys,...)
- Essentially unusual system calls
during deserialization

IMPORTANT: You must respond in EXACTLY this format:
Answer: Malicious
Reason: Brief explanation
...
Example 1:
Log shows: execve("/bin/bash", ["/bin/bash", "-c", "rm
    -rf /"])
Answer: Malicious
Reason: Executes destructive shell command during
    deserialization
...
------------------------------------------------------------
LLM Response:
Answer: Malicious
Reason: The process attempts to connect to a suspicious
    IP address (192.248.1.167) on port 4242, which could
    be a sign of a malicious activity.
```

4.10. Ablation/Probing and Sensitivity Setting

Feature set: All feature sets displayed in Figure 2 were used to study the effect of each feature on the model performance. In this study, the model complexity was varied while keeping the rest of the hyperparameter settings the same as the DYNAHUG (default). The *presence* only model was trained on features representing the presence of individual system calls. The *frequency* only model was trained on frequency of individual system calls. The *presence and frequency* model (DYNAHUG (default)) was trained on a combination of *presence* and *frequency* features. Subsequently, *Sequence* and *Process Sequence* features were concatenated to train the respective *presence, frequency, sequence* and *presence, frequency, sequence, process sequence* models.

Architecture: For this study, *OCSVM*, *SGDSVM* and *IF* were trained with their best hyperparameter setting.

Analysis Type: The *dynamic* model was trained with the workflow showcased in Figure 1. The *static* model processed static opcodes from disassembled models instead of `strace` log data while following the same workflow as the *dynamic* model. The *hybrid* model was trained with the combined feature set of *static* and *dynamic* models.

Clustering: To test whether clustering improves performance, we trained a model without task-based filtering. We then compare this model against DYNAHUG (default).

Dataset Size Sensitivity: To test the sensitivity of our model towards a varying dataset size, we trained our model on the data retrieved from the top `1000`, `2000` (DYNAHUG (default)) and `3000` repositories, sorted by likes.

TABLE 5: DYNAHUG's performance on two (2) task-based clusters

Cluster	Detector	Benign	Malicious		Overall Performance						
		HF	Real	Injected	TP	TN	FP	FN	Precision	Recall	F1-score
text-generation	DYNAHUG	1/2025	25/25	1986/2025	2011/2025	2024/2025	1/2025	14/2025	0.9995	0.9931	0.9963
text-classification	DYNAHUG	28/2004	4/4	1985/2000	1989/2004	1976/2004	28/2004	15/2004	0.9861	0.9925	0.9893

4.11. Implementation Details and Platform

DYNAHUG was implemented in about 4.1k lines of Python code. Experiments and data analysis were implemented in 6k lines of Python code. For crawling, we used the Python SDK for hugging face, i.e., `huggingface_hub` [38]. The downloaded models were then archived for future reference inside a GCS Standard Bucket with the help of the `google-cloud-storage` [63] library. For dynamic analysis, the code was containerized using `Docker` [42]. The PyTorch files were deserialized using `torch` [64] and `strace` [45] was used to observe the runtime behaviour. For the model training phase, `NumPy` [65] and `pandas` [66] libraries were utilized for data analysis, processing and manipulation. Steps related to model training and evaluation were done with the `scikit-learn` [49] machine learning library. `SHAP` [67] was used to explain the impact of the features on which the classifier was trained on. Experiments were setup on n2-highmem-4 (4 vCPUs, 32 GB Memory) Google Cloud Compute Engine instance with Ubuntu, 22.04 LTS Minimal installed. Within this VM, a docker container with a base image of `python:3.10.12-slim` was setup with all project files cloned from the GitHub repository and the required dependencies installed. The LLM baseline experiment was run on Google Colab [68] with a NVIDIA-A100 GPU [60]. HF `transformers` library was utilized to run inference on Llama-3.1. We provide DYNAHUG's source code and our experimental data online to support reproducibility and reuse:

https://dynahug-detector.github.io

5. Results

5.1. RQ1: Effectiveness

In this experiment, we examine the effectiveness of DYNAHUG using two tasks namely, text generation and text classification. Table 5 reports our findings.

We found that DYNAHUG *is effective in detecting malicious PTMs across both clusters*. We observed that DYNAHUG generally performs well on both task clusters across all performance metrics. However, DYNAHUG has a slightly better performance on the text generation cluster than the text classification cluster (0.9963 vs. 0.9893). Across all metrics, DYNAHUG's performance is consistently high (between 0.9861 and 0.9995). All in all, DYNAHUG has a good performance, which is consistently above 0.986 across all metrics and settings. This result implies that the ML-based dynamic analysis technique of DYNAHUG is effective in malicious PTM detection. In summary, our approach (DYNAHUG) generalizes to two different PTM tasks.

> DYNAHUG *is effective ($\geq$ 0.986) in malicious PTM detection across all metrics and clusters.*

5.2. RQ2: State-of-the-art Comparison

This experiment compares DYNAHUG to five open-source detectors including three static detectors (Fickling, ModelScan and PickleScan), and two dynamic detectors, namely ModelTracer, and an LLM-based dynamic detector Llama-3.1. This experiment employs the text generation cluster of HF. Table 6 reports our findings.

We found that DYNAHUG *is up to 44% more effective than the state-of-the-art detectors.* Table 6 shows that DYNAHUG outperforms the closest competitior (ModelTracer) by about 7% (0.9963 vs. 0.9299 F1-score). On the one hand, ModelTracer detects all real malicious PTMs, but it does not generalize on the exact malicious payloads when injected in different models. This demonstrates that its detection is not generalizable to new models. In addition, ModelTracer does not perform well on PyPI injected modules, which underscores its non-exhaustive trace blacklisting. On the other hand, the static detectors have the worst performance, followed by the Llama-3.1-8B-Instruct-tuned. DYNAHUG is 44% (0.9963 vs. 0.6902 F1-score) more effective than the best static detectors, i.e., PickleScan and ModelScan. We attribute the poor performance of the static detectors to the lack of behavioral information, use of rule sets and the over-approximation of static analysis. Finally, DYNAHUG is 33% (0.9963 vs. 0.7483) more effective than Llama-3.1 in malicious PTM detection. We believe the poor performance of Llama-3.1 is due to the lack of PTM-specific security knowledge. Overall, this experiment demonstrates the superiority of DYNAHUG, and the importance of its behavioral analysis and learning approach to malicious PTM detection.

> DYNAHUG *is up to 44% more effective than the state-of-the-art detectors; it is 7% more effective than the closest competitior (ModelTracer).*

5.3. RQ3: Ablation and Probing study

These experiments investigate the contribution of our design decisions to the effectiveness of DYNAHUG. Using the text generation task, we examine the impact of DYNAHUG's dynamic analysis, feature selection and model architecture in comparison to alternative choices.

TABLE 6: Comparison of current open-source SOTA's performance across collected datasets.

Analysis Type	Detector	Benign	Malicious			Overall Performance						
		HF (2025)	Real (25)	Injected (1000)	Injected PyPI (1000)	TP	TN	FP	FN	Precision	Recall	F1-score
Static	PickleScan [20]	0	23	1000	44	1067	2025	0	958	1	0.5269	0.6902
Static	ModelScan [21]	0	23	1000	44	1067	2025	0	958	1	0.5269	0.6902
Static	Fickling [24]	2025	25	1000	1000	2025	0	2025	0	0.5	1	0.6667
Dynamic	ModelTracer [23]	0	25	828	907	1760	2025	0	265	1	0.8691	0.9299
Dynamic	Llama-3.1 [57]	1337	22	993	995	2010	688	1337	15	0.6005	0.9926	0.7483
Dynamic	DynaHug (default)	1	25	1000	986	2011	2024	1	14	0.9995	0.9931	0.9963

TABLE 7: DynaHug's performance under different analysis types (best results are in **bold** text)

Analysis Type	Detector	Benign	Malicious		Overall Performance						
		HF (2025)	Real (25)	Injected (2000)	TP	TN	FP	FN	Precision	Recall	F1-score
Static	DynaHug	2025	**25**	**2000**	**2025**	0	2025	**0**	0.5000	**1.0000**	0.6667
Dynamic	DynaHug (default)	**1**	**25**	1986	2011	**2024**	**1**	14	**0.9995**	0.9931	**0.9963**
Hybrid	DynaHug	**1**	**25**	1986	2011	**2024**	**1**	14	**0.9995**	0.9931	**0.9963**

TABLE 8: DynaHug's effectiveness with varying feature sets showing 'freq" = frequency, "seq" = sequence and "proc" = process (best results are in **bold** text)

Feature Set	Detector	Benign	Malicious		Overall Performance						
		HF (2025)	Real (25)	Injected (2000)	TP	TN	FP	FN	Precision	Recall	F1-score
presence	DynaHug	0	22	1664	1686	**2025**	0	339	**1.0000**	0.8326	0.9086
freq	DynaHug	**1**	**25**	1986	**2011**	**2024**	**1**	14	0.9995	0.9931	**0.9963**
presence, freq	DynaHug (default)	**1**	**25**	1986	**2011**	**2024**	**1**	14	0.9995	0.9931	**0.9963**
presence, freq, seq	DynaHug	33	24	**2000**	**2024**	1992	33	**1**	0.9840	**0.9995**	0.9917
presence, freq, seq, proc seq	DynaHug	51	24	**2000**	**2024**	1974	51	**1**	0.9754	**0.9995**	0.9873

Analysis Type: This experiment examines the effectiveness of DynaHug's dynamic analysis in comparison to static analysis and hybrid analysis. Table 7 reports our findings.

We found that DynaHug *with dynamic analysis is more effective than* DynaHug *with static analysis.* Table 7 shows that dynamic analysis in DynaHug is twice (2x) as effective as static analysis (precision of 0.5 vs. 0.9995). Meanwhile, hybrid analysis has the same performance as dynamic analysis. We observe that static analysis is not as effective as dynamic analysis. Besides, combining dynamic and static analyses does not improve the performance of DynaHug, it is as effective as using *only* dynamic analysis (*see* Table 7). We attribute the weak performance of static analysis to the lack of precise execution information which causes over-approximation. This result also aligns with the performance of the static detectors in **RQ2** (*see* Table 6). Overall, this result shows that DynaHug's use of dynamic analysis contributes positively to its detection effectiveness.

Dynamic analysis contributes positively to detection effectiveness of DynaHug*; it is twice as effective as using only static analysis.*

Feature Selection: We examine the effectiveness of the default setting of DynaHug versus smaller and larger set of features. We report our findings in Table 8.

Results show that *the default feature setting of* DynaHug *(presence and frequency of system calls) contributes positively to its effectiveness and outperforms alternative features setting.* More importantly, Table 8 shows that the *frequency of system calls* improves DynaHug's detection effectiveness more than all other feature settings. In comparison to presence only features, the frequency of system calls improves the recall and F1-score of DynaHug by up to 19% (0.8326 vs. 0.9931) and ≈10% (0.9086 vs. 0.9963), respectively. Meanwhile, employing a larger feature set than the default settings (e.g., adding sequence of system calls and/or sequence of system processes) leads to poorer performance with a reduced the precision and F1-score (*see* Table 8). We attribute the performance of DynaHug's *default* setting to its use of the *frequency of system calls* which makes DynaHug detect outlier system call distribution. This observation aligns with our explainability findings. For instance, Figure 3 shows that the frequency of `set_robust_list, clone and futex` calls are the most important features for predicting malicious models (*see* Figure 3(b)). These results show that DynaHug's feature setting contributes positively to its detection effectiveness.

The default feature setting of DynaHug *outperforms alternative feature settings and the frequency of system calls contributes the most to its performance.*

ML Architecture: We compare DynaHug's ML architecture (`OCSVM`)) to alternative ML architectures for novelty/anomaly detection namely, *Isolation Forest* (`IF`) and *SGD One Class SVM* (`SGDOCSVM`). Table 9 reports our results.

We observed that *the default model architecture in* DynaHug *(`OCSVM`) outperforms the tested alternative architectures (`IF` and `SGDOCSVM`).* Table 9 shows that default DynaHug (OCSVM) performs best in terms of precision and F1-score: DynaHug (`OCSVM`) performs best (F1 = 0.9963), followed by DynaHug with `SGDOCSVM` (F1 = 0.9419), then DynaHug with `IF` (F1 = 0.9837). However, we observed that DynaHug with `SGDOCSVM` has a slightly better recall (1.0 vs. 0.9931). We attribute the better performance of the the default DynaHug (`OCSVM`) to the size of our dataset, since the tested alternatives (`IF` and `SGDOCSVM`) are often sensitive to the size of the training dataset and more suitable for training larger datasets

TABLE 9: DYNAHUG's performance across different architectures (best results are in **bold** text)

Architecture	Detector	Benign	Malicious		Overall Performance						
		HF (2025)	Real (25)	Injected (2000)	TP	TN	FP	FN	Precision	Recall	F1-score
IsolationForest	DYNAHUG	4	24	1782	1806	2021	4	219	0.9978	0.8919	0.9419
OneClassSVM	DYNAHUG (default)	**1**	**25**	1986	**2011**	**2024**	**1**	14	**0.9995**	0.9931	**0.9963**
SGDOneClassSVM	DYNAHUG	67	**25**	**2000**	**2025**	1958	67	**0**	0.9680	**1.000**	0.9837

TABLE 10: Impact of clustering on DYNAHUG (best results are in **bold** text)

Cluster	Detector	Benign Set	Malicious Set	Overall Performance						
				TP	TN	FP	FN	Precision	Recall	F1-score
text-generation	DYNAHUG	**4/200**	198/200	198/200	**196/200**	**4/200**	2/200	**0.9802**	0.9900	**0.9851**
text-classification	DYNAHUG	7/200	**199/200**	**199/200**	193/200	7/200	**1/200**	0.9660	**0.9950**	0.9803
non-clustered	DYNAHUG	9/200	**199/200**	**199/200**	191/200	9/200	**1/200**	0.9567	**0.9950**	0.9754

[69, 70, 71]. In particular, `OCSVM` is more suitable for DYNAHUG due to DYNAHUG's task clustering step (causing a reduced training size) and the huge computational cost of training larger datasets. This result demonstrates that DYNAHUG's architecture (`OCSVM`) fits its design and outperforms close alternatives (DYNAHUG with `IF` or `SGDSVM`).

> DYNAHUG*'s ML architecture (`OCSVM`) fits its design and outperforms its close alternatives (`IF` or `SGDSVM`).*

5.4. RQ4: Probing and Sensitivity Study

We conduct probing studies examining our design decisions for DYNAHUG's model training. In particular, we examine the impact of clustering and varying training dataset size on DYNAHUG's performance. Since these experiments informed our selection of the best DYNAHUG setting for all other evaluation, it was conducted during training setup using *only* the test set without contamination with the evaluation sets in **RQ1** to **RQ3**, for a fair and balanced analysis. Table 10 and Table 11 report our findings.

Task Clustering: During DYNAHUG's design we examine the effectiveness of clustering on task-based tags by training DYNAHUG on two clusters and a non-clustered set of PTMs. Table 10 reports the performance of DYNAHUG.

We found that DYNAHUG *with clustering outperforms* DYNAHUG *without clustering*. Table 10 shows that DYNAHUG performs better when it is trained on the text classification or text generation clusters versus on the Top-2000 models on HF (without clustering). This result holds across all metrics. For instance, the F1-score of the DYNAHUG trained on the text generation cluster is 0.9851, while the non-clustered DYNAHUG has an F1-score of 0.9754. We attribute the performance of the clustered setting to the fact that it learns the specific behavior of a particular task better than the non-clustered model. This finding demonstrates the importance of clustering and motivates our design decision to cluster before training.

> *Task-based clustering leads to improved performance relative to a non-clustered model.*

Sensitivity to training dataset size: We examine the sensitivity of DYNAHUG to different dataset sizes. Using the text generation task, we train two additional models, with 1000 PTMs and 3000 PTMs. We report our findings in Table 11.

We found that *the effectiveness of* DYNAHUG *slightly improves as the training dataset increases*. Table 11 shows that training on a smaller dataset (1000) has a lower F1-score than training on the default DYNAHUG setting (2000), and training on even more PTMs (3000) outperforms the default DYNAHUG setting. DYNAHUG's performance is sensitive to the size of the dataset. We attribute the performance of larger datasets to the contribution of additional data points for generalizing the training. However, there is a trade-off between analysis cost and improved performance. Thus, we employ the 2,000 training data size for all other experiments, since the performance improvement with 3,000 PTMs is not significant (0.9884 vs. 0.9851) and it is computationally expensive to collect dynamic analysis on a larger set of models across each study, (e.g., thousands of models for additional settings in the ablation studies (RQ3)).

> DYNAHUG*'s effectiveness slightly improves as the size of the training dataset increases.*

6. Limitations and Threats to Validity

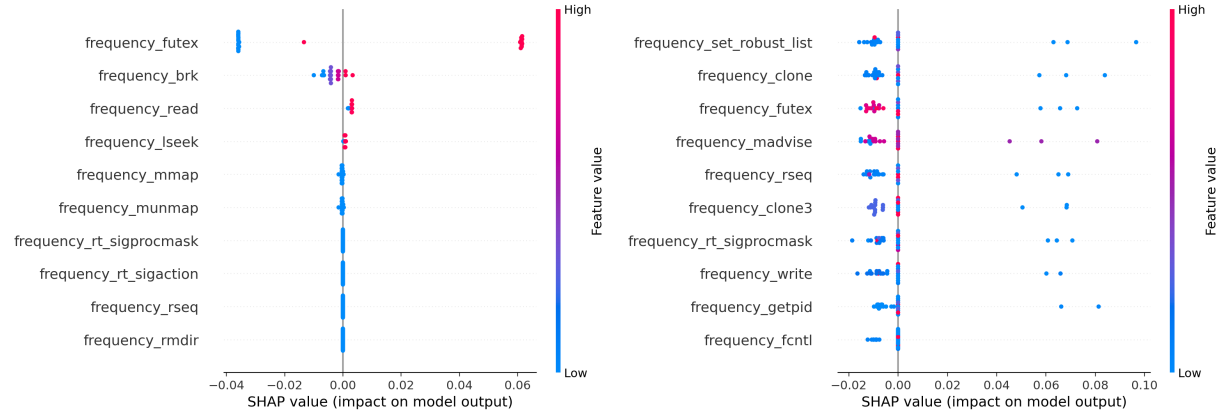
Internal Validity: The main threats to internal validity of this work and DYNAHUG are the correctness of our implementation and the reproducibility of our results. To mitigate this threat, we have tested our implementation and resulting models under varying settings to ensure reproducibility (*see* **RQ3**). We also provide our source code, models and experimental data to support scrutiny and reuse.

External Validity: The main risk to construct validity is the generalizability of DYNAHUG and our findings. In particular, our findings may not generalize to other Model Hubs, tasks or unpopular (non-Top-K) models. To mitigate this threat, we have performed our experiments using the most popular model hub (Hugging Face), tested on two popular clusters and thousands of benign and malicious models. We have also tested on different parameter settings in our ablation and sensitivity study (**RQ3** and **RQ4**). However, despite best efforts, we acknowledge that our findings may not generalize to other settings, beyond the tested settting.

Construct Validity: To avoid experimenter bias in this work, we have employed thousands of models from Hugging

TABLE 11: DYNAHUG's sensitivity to varying dataset size (best results are in **bold** text)

Training Size	Detector	Benign Set	Malicious Set	Overall Performance						
				TP	TN	FP	FN	Precision	Recall	F1-score
1000	DYNAHUG	4/100	**100/100**	**100/100**	96/100	4/100	**0/100**	0.9615	**1.0000**	0.9804
2000	DYNAHUG (default)	4/200	198/200	198/200	196/200	4/200	2/200	0.9802	0.9900	0.9851
3000	DYNAHUG	**4/300**	297/300	297/300	**296/300**	**4/300**	3/300	**0.9867**	0.9900	**0.9884**



(a) DYNAHUG (default) on Benign dataset

(b) DYNAHUG (default) on Malicious dataset

Figure 3: Top-10 important features that explain DYNAHUG's prediction on the benign and malicious datasets using SHAP.

Face and related scanners (e.g., MALHUG). We have also compared our approach to five (5) open-source state-of-the-art detectors and filtered our training set using the scanners provided by Hugging Face security API. Finally, we have also manually inspected the real-world malicious PTMs to confirm they are malicious and constructed test oracles to check that our injector indeed inject malicious payloads.

7. Related Works

Static Detectors: These tools aim to detect malicious PTMs by inspecting its code for potential malicious behaviors, e.g., using disassemblers like pickletools [25] or Fickling [25]. Detection is typically performed through a blacklist of unsafe opcodes (e.g., PickleScan, Modelscan, HF PickleScan [18, 20, 21]) or a whitelist of safe opcodes (e.g., PickleBall [23]). To further overcome the limits of a static list of rules, some static detectors also employ methods such as heuristics (MALHUG [8]), policy generation (PickleBall [25]) or data flow analysis (Fickling [24]). Unlike these approaches, DYNAHUG employs dynamic behavior of PTMs rather than static analysis and relies on ML rather than static blacklisting or whitelisting.

Dynamic Detectors: Similar to DYNAHUG, ModelTracer [23] detects malicious PTMs using dynamic analysis, in particular, strace [45] and Python's sys [72]. It processes the generated syscall data and then checks for sensitive syscall presence like `exec`, `connect` and `chmod` as indicators of malicious behaviour. Invariably, it employs a blacklist of system calls to detect malicious PTMs. Unlike ModelTracer, DYNAHUG relies on the ML classifier to learn the behaviours of benign models, rather than relying on a list of sensitive calls that are rigid, can be easily bypassed and require expert review for edge cases.

8. Conclusion

This work proposes a novel method for detecting malicious PTM models. The goal of our work is to ensure that PTM users can detect malicious PTM models provided by third-party vendors in the ML supply chain system. In particular, we aim to ensure the safety of PTMs uploaded on Model Hubs like Hugging Face or executed in trusted user environments. Our approach (DYNAHUG) employs a combination of dynamic analysis, task-specific clustering and ML to detect malicious PTM models. We orchestrate DYNAHUG by employing a one-class SVM to learn the behavior of the top-K (2000) most liked models in a task-specific cluster (e.g., text generation) of Hugging Face. We also evaluate DYNAHUG using over 18,000 benign and malicious models from different sources, including Hugging Face, MALHUG and injected malicious payloads using MALHUG and PyPI modules. Besides, we compare the effectiveness of DYNAHUG to five baseline detectors, including static, dynamic and LLM-based detectors. Our evaluation results demonstrate that DYNAHUG is effective in detecting malicious PTMs, outperforms the baselines. This work motivates the need to employ behavioral analysis for PTM detection and reduce reliance on blacklists or whitelists. In the future, we plan to expand our study to other Model Hubs and task clusters, and deploy it for broad analysis in the wild.

To support reproducibility and reuse, we provide DYNAHUG's source code and our experimental data:

https://dynahug-detector.github.io/

References

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

[2] "Kaggle: Your Machine Learning and Data Science C ommunity," https://www.kaggle.com/, [Accessed 07-11-2025].

[3] "GitHub," https://github.com/, [Accessed 07-11-2025].

[4] "OpenCSG," https://opencsg.com/, [Accessed 07-11-2025].

[5] "Spark NLP," https://sparknlp.org/, [Accessed 07-11-2025].

[6] "ModelScope," https://modelscope.cn/home, [Accessed 07-11-2025].

[7] Ronik, 2024. [Online]. Available: https://weam.ai/blog/guide/huggingface-statistics/

[8] J. Zhao, S. Wang, Y. Zhao, X. Hou, K. Wang, P. Gao, Y. Zhang, C. Wei, and H. Wang, "Models are codes: Towards measuring malicious code poisoning attacks on pre-trained model hubs," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. ACM, Oct. 2024, p. 2087–2098. [Online]. Available: http://dx.doi.org/10.1145/3691620.3695271

[9] T. H. News, "New Hugging Face Vulnerability Exposes AI Models to Supply Chain Attacks," https://thehackernews.com/2024/02/new-hugging-face-vulnerability-exposes.html, [Accessed 09-11-2025].

[10] ——, "Over 100 Malicious AI/ML Models Found on Hugging F ace Platform," https://thehackernews.com/2024/03/over-100-malicious-aiml-models-found-on.html, [Accessed 09-11-2025].

[11] ——, "Malicious ML Models on Hugging Face Leverage Broken Pickle Format to Evade Detection," https://thehackernews.com/2025/02/malicious-ml-models-found-on-hugging.html, [Accessed 09-11-2025].

[12] E. Montalbano, https://www.darkreading.com/application-security/hugging-face-ai-platform-100-malicious-code-execution-models, Feb 2024.

[13] K. Poireault, "Malicious AI Models on Hugging Face Exploit Novel A ttack Technique," https://www.infosecurity-magazine.com/news/malicious-ai-models-hugging-face/, [Accessed 09-11-2025].

[14] "Malicious ML models discovered on Hugging Face platform — ReversingLabs," https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face, [Accessed 09-11-2025].

[15] "Federal Register :: Request Access," https://www.federalregister.gov/documents/2023/11/01/2023-24283/safe-secure-and-trustworthy-development-and-use-of-artificial-intelligence, [Accessed 09-11-2025].

[16] "cve.org," https://www.cve.org/CVERecord/SearchResults?query=pickle, [Accessed 09-11-2025].

[17] H. Face, "Third-party scanner: Protect ai," https://huggingface.co/docs/hub/en/security-protectai, 2025, hugging Face documentation, accessed: 2025-10-11.

[18] ——, "Pickle scanning (hub documentation)," https://huggingface.co/docs/hub/en/security-pickle#what-we-have-now, 2025, hugging Face documentation, accessed: 2025-10-11.

[19] ——, "Third-party scanner: Jfrog," https://huggingface.co/docs/hub/en/security-jfrog, 2025, hugging Face documentation, accessed: 2025-10-11.

[20] mmaitre314, "picklescan: Security scanner detecting python pickle files performing suspicious actions," https://github.com/mmaitre314/picklescan, 2025, gitHub repository, accessed: 2025-10-11.

[21] P. AI, "Modelscan: Protection against model serialization attacks," https://github.com/protectai/modelscan, 2025, gitHub repository, accessed: 2025-10-11.

[22] "Client Challenge," https://pypi.org/project/execute/, [Accessed 10-11-2025].

[23] B. Casey, J. C. S. Santos, and M. Mirakhorli, "A large-scale exploit instrumentation study of ai/ml supply chain attacks in hugging face models," 2024. [Online]. Available: https://arxiv.org/abs/2410.04490

[24] T. of Bits, "Fickling: A python pickling decompiler and static analyzer," https://github.com/trailofbits/fickling, 2025, gitHub repository, accessed: 2025-10-11.

[25] A. D. Kellas, N. Christou, W. Jiang, P. Li, L. Simon, Y. David, V. P. Kemerlis, J. C. Davis, and J. Yang, "Pickleball: Secure deserialization of pickle-based machine learning models (extended report)," 2025. [Online]. Available: https://arxiv.org/abs/2508.15987

[26] P. Contributors, "weights_only_unpickler.py – pytorch," https://github.com/pytorch/pytorch/blob/main/torch/_weights_only_unpickler.py, 2025, github repository, Accessed: 2025-11-10.

[27] "llm-stacking/G_learn_depth at main — huggingface.co," https://huggingface.co/llm-stacking/G_learn_depth/tree/main, [Accessed 11-11-2025].

[28] "Narsil/totallysafe at main — huggingface.co," https://huggingface.co/Narsil/totallysafe/tree/main, [Accessed 11-11-2025].

[29] Zollllll, ""dont_download_this" – model card," https://huggingface.co/Zollllll/dont_download_this, 2025, accessed: 2025-11-10.

[30] "pytorch/Phi-4-mini-instruct-FP8 at main — huggingface.co," https://huggingface.co/pytorch/Phi-4-mini-instruct-FP8/tree/main, [Accessed 10-11-2025].

[31] F. Hijazi, ""totally-harmless-model" – model card," https://huggingface.co/FarisHijazi/totally-harmless-model, 2025, accessed: 2025-11-10.

[32] "ankushvangari-org2/unsafe_model · Hugging Face," https://huggingface.co/ankushvangari-org2/unsafe_model, [Accessed 09-11-2025].

[33] "star23/round2," https://huggingface.co/star23/round2/tree/main, [Accessed 09-11-2025].

[34] "Client Challenge — pypi.org," https://pypi.org/search/?q=execute, [Accessed 11-11-2025].

[35] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1025–1035.

[36] T. S. R. Pimenta, F. Ceschin, and A. Gregio, "Androidgyny: Reviewing clustering techniques for android malware family classification," *Digital Threats*, vol. 5, no. 1, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3587471

[37] D. George, A. Mauldin, J. Mitchell, S. Mohammed, and R. Slater, "Static malware family clustering via structural and functional characteristics," *SMU Data Science Review*, vol. 7, no. 2, p. 4, 2023.

[38] "Hub client library — huggingface.co," https://huggingface.co/docs/huggingface_hub/index, [Accessed 07-11-2025].

[39] D. Cohen, "Data Scientists Targeted by Malicious Hugging Face ML Models with Silent Backdoor — jfrog.com," https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/, 2024, [Accessed 27-10-2025].

[40] "huggingface/transformers github," https://github.com/huggingface/transformers/blob/main/src/transformers/modeling_utils.py#L371, [Accessed 13-10-2025].

[41] "Cloud Storage," https://cloud.google.com/storage?hl=en, [Accessed 11-11-2025].

[42] I. Docker, "Docker: Accelerated Container Application Development — docker.com," https://www.docker.com/, [Accessed 06-11-2025].

[43] "What is a Container? — Docker — docker.com," https://www.docker.com/resources/what-container/, [Accessed 11-11-2025].

[44] Canonical, "Strace man page." [Online]. Available: https://manpages.ubuntu.com/manpages/jammy/man1/strace

[45] "strace," https://github.com/strace/strace, [Accessed 12-10-2025].

[46] "TCPDUMP; LIBPCAP," https://www.tcpdump.org/, [Accessed 12-10-2025].

[47] "eBPF - Introduction, Tutorials & Community Resources — ebpf.io," https://ebpf.io/, [Accessed 12-10-2025].

[48] F. PACULL, "Loading data into a Pandas DataFrame – a performance study," https://www.architecture-performance.fr/ap_blog/loading-data-into-a-pandas-dataframe-a-performance-study/, 2019, [Accessed 14-10-2025].

[49] "User Guide — scikit-learn.org," https://scikit-learn.org/stable/user_guide.html, [Accessed 07-11-2025].

[50] PyPI, "PyPI &xB7; The Python Package Index — pypi.org," https://pypi.org, [Accessed 10-11-2025].

[51] Nmap Project, "Ncat — nmap's netcat replacement," https://nmap.org/ncat/, 2025, accessed: 2025-11-10.

[52] "gabejabe/bsidesSF-gordon-ramsey · Hugging Face — huggingface.co," https://huggingface.co/gabejabe/bsidesSF-gordon-ramsey, [Accessed 11-11-2025].

[53] "syscalls(2) - Linux manual page — man7.org," https://man7.org/linux/man-pages/man2/syscalls.2.html#:~:text=/usr/include/asm/unistd.h, [Accessed 30-10-2025].

[54] P. Kassianik, B. Saglam, A. Chen, B. Nelson, A. Vellore, M. Aufiero, F. Burch, D. Kedia, A. Zohary, S. Weerawardhena *et al.*, "Llama-3.1-foundationai-securityllm-base-8b technical report," *arXiv e-prints*, pp. arXiv–2504, 2025.

[55] "Meta Releases LlamaFirewall, an Open-Source Defense Against AI Hijacking — deeplearning.ai," https://www.deeplearning.ai/the-batch/meta-releases-llamafirewall-an-open-source-defense-against-ai-hijacking/, [Accessed 12-11-2025].

[56] C. Rondanini, B. Carminati, E. Ferrari, A. Kundu, and A. Gaudiano, "Malware detection at the edge with lightweight llms: A performance evaluation," *ACM Transactions on Internet Technology*.

[57] Meta, "meta-llama/Llama-3.1-8B-Instruct," https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct, [Accessed 09-11-2025].

[58] "GitHub - s2e-lab/hf-model-analyzer — github.com," https://github.com/s2e-lab/hf-model-analyzer.git, [Accessed 11-11-2025].

[59] "Llama 3.1 Release," https://ai.meta.com/blog/meta-llama-3-1/, [Accessed 09-11-2025].

[60] C. Perry, "Colab Nvidia GPU," https://blog.tensorflow.org/2022/09/colabs-pay-as-you-go-offers-more-access-to-powerful-nvidia-compute-for-machine-learning.html, 2022, [Accessed 13-11-2025].

[61] P. Brown, A. Brown, M. Gupta, and M. Abdelsalam, "Online malware classification with system-wide system calls in cloud iaas," in *2022 IEEE 23rd International Conference on Information Reuse and Integration for Data Science (IRI)*. IEEE, 2022, pp. 146–151.

[62] "What is few shot prompting?" https://www.ibm.com/think/topics/few-shot-prompting, [Accessed 11-11-2025].

[63] "Python client libraries — Google Cloud Documentation," https://docs.cloud.google.com/python/docs/reference/storage/latest, [Accessed 11-11-2025].

[64] "PyTorch documentation; PyTorch 2.9 documentation — docs.pytorch.org," https://docs.pytorch.org/docs/stable/index.html, [Accessed 07-11-2025].

[65] "NumPy documentation; NumPy v2.3 Manual — numpy.org," https://numpy.org/doc/stable/, [Accessed 07-11-2025].

[66] "pandas documentation; pandas 2.3.3 documentation — pandas.pydata.org," https://pandas.pydata.org/docs/, [Accessed 07-11-2025].

[67] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions." Curran Associates, Inc., 2017. [Online]. Available: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[68] "colab.google," https://colab.google/, [Accessed 14-11-

2025].
[69] C. A. Ramezan, T. A. Warner, A. E. Maxwell, and B. S. Price, "Effects of training set size on supervised machine-learning land-cover classification of large-area high-resolution remotely sensed data," *Remote Sens. (Basel)*, vol. 13, no. 3, p. 368, Jan. 2021.
[70] D. Rajput, W.-J. Wang, and C.-C. Chen, "Evaluation of a decided sample size in machine learning applications," *BMC Bioinformatics*, vol. 24, no. 1, p. 48, Feb. 2023.
[71] J. Prusa, T. M. Khoshgoftaar, and N. Seliya, "The effect of dataset size on training tweet sentiment classifiers," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 2015, pp. 96–102.
[72] P. Developers, "`CPython`: The python programming language," https://github.com/python/cpython, gitHub repository, accessed 2025-10-12.