PickleBall: Secure Deserialization of Pickle-based Machine Learning Models

Andreas D. Kellas
Columbia University
New York, NY, USA
andreas.kellas@columbia.edu

Neophytos Christou
Brown University
Providence, RI, USA
neophytos_christou@brown.edu

Wenxin Jiang*
Purdue University
West Lafayette, IN, USA
jiang784@purdue.edu

Penghui Li
Columbia University
New York, NY, USA
pl2689@columbia.edu

Laurent Simon
Google
Mountain View, CA, USA
laurentsimon@google.com

Yaniv David
Technion
Haifa, Israel
yanivmd@technion.ac.il

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

James C. Davis
Purdue University
West Lafayette, IN, USA
davisjam@purdue.edu

Junfeng Yang
Columbia University
New York, NY, USA
junfeng@cs.columbia.edu

Abstract

Machine learning model repositories, such as the Hugging Face Model Hub, facilitate model exchanges. However, bad actors can deliver malware through compromised models. Existing defenses, such as safer model formats, restrictive (but inflexible) loading policies, and model scanners, have shortcomings: 44.9% of popular models on Hugging Face still use the insecure pickle format, 15% of these cannot be loaded by restrictive loading policies, and model scanners have both false positives and false negatives. Pickle remains the *de facto* standard for model exchange, and the ML community lacks a tool that offers transparent safe loading.

We present PickleBall to help machine learning engineers load pickle-based models safely. PickleBall statically analyzes the source code of machine learning libraries and computes custom policies that specify a safe load-time behavior for benign models. It then dynamically enforces these policies during load time as a drop-in replacement for the pickle module. PickleBall generates policies that correctly load 79.8% of benign pickle-based models in our dataset, while rejecting all (100%) malicious examples in the same dataset. In comparison, evaluated model scanners fail to identify known malicious models, and the state-of-the-art loader loads 22% fewer benign models than PickleBall. PickleBall removes the threat of arbitrary function invocation from malicious pickle-based models, raising the bar for attackers as they have to depend on code reuse techniques.

*Also with Socket.

CCS Concepts

• **Security and privacy → Software and application security**; **Malware and its mitigation**.

Keywords

Secure Model Loading; Deserialization Attacks; Supply Chains

ACM Reference Format:
Andreas D. Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P. Kemerlis, James C. Davis, and Junfeng Yang. 2025. PickleBall: Secure Deserialization of Pickle-based Machine Learning Models. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765037

1 Introduction

Open-source and open-weight models enable the AI ecosystem [26, 39, 52, 98]. They allow machine learning engineers to exchange pre-trained models rather than training from scratch [29], facilitating direct use or fine-tuning [14]. Open model repositories like Hugging Face now host millions of pre-trained models for many tasks [50, 52]. These model hubs are accessed directly as well as through corporate mirrors [107], with billions of downloads per month.

Similar to traditional software supply-chain attacks, bad actors can distribute malicious models. The most common strategy for achieving remote code execution is tampering with the model deserialization process. Several model serialization formats, such as TorchScript [78], H5/HDF5 [90], and the Python pickle module [75], permit executable metadata or callbacks during model deserialization. Attackers can craft malicious serialized models to execute code, such as `os.system()`, on victim systems during model loading [12, 16, 23, 100]. Researchers have found malicious pickle models on Hugging Face whose payloads include system fingerprinting, credential theft, and reverse shells [16, 100, 107], with one study discovering a 5× increase in the rate of malicious models uploaded to Hugging Face year-over-year [107].

This led to alternative safe model formats like SafeTensors [42], and restricted loading APIs like the PyTorch weights-only unpickler [79]; we study their adoption in the Hugging Face ecosystem (§3) and find that nevertheless, insecure formats are still prevalent.

In this work, we propose a novel approach to secure pickle model deserialization, which we focus on for three reasons. First, pickle is a popular exchange format for models. Repositories with pickle models are downloaded over 2.1 billion times per month from the Hugging Face model hub (§3.1). Second, pickle is the most expressive format and thus is challenging to secure. Models are encoded as opcodes that are executed by the pickle virtual machine [40, 67] during deserialization, which permits invocations of arbitrary Python classes and functions (*callables*). Third, pickle is abused by attackers. Almost all malicious models on Hugging Face use the pickle format.

Our evaluation shows that the two existing kinds of pickle deserialization defenses are inadequate. *Model scanners* [7, 12, 41, 107] maintain fixed denylists of disallowed callables to identify models that invoke them. *Safe model loaders* [79] use fixed allowlists to permit only the use of trusted callables. Our evaluation shows the limits of these inflexible approaches for ML models. For instance, the default safe deserialization loader in PyTorch [79] prevents 15% of Hugging Face pickle repositories from loading (§3.1).

To address these limitations, we present PickleBall, a two-part system for securing the exchange of pickle-based models. Our insight is that we can analyze library code to determine the expected behaviors of benign models produced by the library, and enforce *tailored model loading policies*. PickleBall statically analyzes the library code to learn (1) all class types used in the library's models and their transitive attribute types, and (2) all functions needed to restore objects of these types. Then, PickleBall's model-loading component enforces the inferred policies.

We evaluated PickleBall and the state-of-the-art approaches on a dataset of 336 models. Our dataset is meant to represent the kinds of models that PickleBall must handle, including malicious and benign models. We used 252 benign models sourced from Hugging Face, and 84 real and synthetic malicious models. PickleBall loads 79.8% of benign models and prevents all malicious models from executing their payloads. PickleBall adds a runtime overhead of ~2.62% to safely load a model. Compared to other approaches, PickleBall achieved favorable precision and recall.

In summary, we contribute:

(1) An **ecosystem-scale study** of pickle security considerations in the Hugging Face Model Hub. Repositories with *only* pickle models are downloaded over 400 million times per month, despite the introduction of new model formats. We find that 15% of repositories with only pickle models have a model that cannot be loaded by the SOTA secure pickle model loader, the weights-only unpickler.
(2) The **design and implementation of PickleBall**, a framework for securely loading pickle models. PickleBall tailors loading policies to models, and enforces these policies lazily, for secure, efficient, and robust model loading.[1]
(3) A **novel dataset** of 336 benign and malicious pickle-based models for evaluating pickle security efforts. It has 252 benign models collected from Hugging Face, and 84 malicious models.

[1]PickleBall is available at https://github.com/columbia/pickleball.

2 Background

Here we describe the ML model reuse ecosystem and formats (§2.1), then how the pickle format is used and the risks it entails (§2.2).

2.1 Model Reuse

2.1.1 The Model Supply Chain. Training models from scratch requires significant resources [29, 72], so engineers and companies reuse machine-learning models trained by others (pre-trained models) [26, 39, 52, 98]. Open model repositories like Hugging Face host over 1.8M models [44] for many tasks [50, 52].

A supply chain of pre-trained models has grown from model reuse, which comes with risks similar to those in the traditional software supply chain [52, 68]. Bad actors apply techniques familiar in traditional software security, such as typosquatting [48, 49, 58, 65] and code injection [99], as well as ML-specific techniques like model and data manipulation [38]. Model hubs like Hugging Face try to detect malicious models using both traditional code scanners like ClamAV [52] and domain-specific pickle scanners [7, 41] due to the proliferation of malicious pickle models [12, 107].

Machine learning libraries facilitate the development and exchange of models. Libraries like PyTorch [70], TensorFlow [6], and JAX [10] provide a core of general ML library utilities, like model training and serialization functions. Other popular but more specific libraries build upon the core libraries with task-specific utilities, like ultralytics (formerly YOLO) [96] for image recognition, PyAnnote [74] for audio processing, and flair [66] for text processing. To create a model, an engineer uses one of these libraries to write a *model saver* program, which trains a model and serializes it for reuse. To load and use the model, an engineer uses the same library (identified in documentation that accompanies the model) to write a *model loader* program. The libraries provide the interface for interacting with the shared models.

2.1.2 Model Serialization Formats. Model savers and loaders must agree on the serialization format; there are various formats available, each with its own tradeoffs in terms of security, flexibility, and performance.[2] Python is the primary language for using ML models, and its native serialization module, pickle [75], provides a convenient and flexible interface for saving objects; pickle proliferated for being easy to use and is used by popular libraries like PyTorch [77]. Hugging Face released the SafeTensors format in September 2022 as an alternative that prioritizes security [42]. The GGUF format, released August 2023, is optimized for fast model loading and inference tasks, especially for large language models [36, 43]. Other formats may be selected for library coupling (e.g., TensorFlow SavedModel), interoperability (e.g., ONNX), or intermediate tradeoffs between security, flexibility, and performance.

The security of a format depends on the expressivity of its operations. The SafeTensors format requires very few different operations to load a model, because it only encodes model weight values, and is considered a secure format after independent security audits [71, 91]. Some formats, like TensorFlow SavedModel and ONNX, are known to permit operations that could be abused in specific settings [92, 108], but with no observed real-world attacks.

[2]A table showing these tradeoffs is provided in the SafeTensors repository README: https://github.com/huggingface/safetensors.

```
1  import pickle
2
3  def read_weights_to_tensor(filename: str) -> Tensor:
4      # Read a file containing weights and
5      # return a Tensor object.
6
7  class Tensor(object):
8      ...
9      def __reduce__(self):
10         return (read_weights_to_tensor, (self.filename,))
11
12 class Model(object):
13     def __init__(self, weights: library.Tensor):
14         self.weights = weights
15
16     def save(self, filename):
17         with open(filename, 'wb') as fd:
18             pickle.dump(self, fd)
19
20     @classmethod
21     def load(cls, path):
22         with open(path, 'rb') as fd:
23             return pickle.load(fd)
24     ...
```

Figure 1: Example of a machine learning library with a model that can be pickled. The `Tensor` class's `__reduce__` method registers the `read_weights_to_tensor` function for execution during deserialization.

Pickle is an extremely expressive format that permits nearly arbitrary operations during deserialization, and numerous malicious pickle models are discovered on Hugging Face [12, 16, 100, 104, 107]. We focus our efforts on pickle models due to their insecure format (cf. §2.2) and their continued popularity (cf. §3).

2.2 Pickle Serialization and Risks

Pickle is popular because of its flexibility and convenience, due to its ability to represent almost any Python object. The pickle module implements a virtual machine, the *Pickle Machine (PM)*, that executes a sequence of opcodes [53] to deserialize an object. The expressiveness of the PM allows it to serialize and reconstruct complex Python data structures, but also make it vulnerable to attacks, allowing attackers to invoke arbitrary Python callables [67].

Pickle Program Structure and Semantics: A pickle program consists of opcode sequences interpreted by the PM, a stack-based VM implemented in the Python pickle module [67]. When the pickle program halts, the object at the top of the PM stack is returned to the Python interpreter as the deserialized object.

The PM is integrated into the Python interpreter. Many of the PM's opcodes create or manipulate native-type objects, e.g., `NEWFALSE` to create a `bool`, while others import and invoke Python *callables* (classes and functions) [76]. Specifically, the *callable importing* opcodes `GLOBAL` and `STACK_GLOBAL` take a callable's name, import it, and push it to the top of the PM stack. Class instances are instantiated using *callable allocating* opcodes, like `NEWOBJ`, which calls the class's `__new__` method. Function references are called via *callable invoking* opcodes like `REDUCE`. Arguments can be passed to both allocations and invocations, and the return value is pushed to the PM stack. Lastly, the *callable building* `BUILD` opcode can modify an object (e.g., set/change its attributes). Pickle lets users customize the deserialization process with the `__reduce__` method.

The method must return a reference to a function and arguments. During serialization, an object's `__reduce__` method is called and the returned function reference and argument values are written to the serialized output opcodes. During deserialization, the function is invoked with the arguments, and the return value is pushed onto the PM stack. This provides a primitive to invoke arbitrary functions in a pickle program. Figure 1 shows an example class that registers a handler function (line 10) that is invoked with a *callable invoking* opcode during deserialization.

Pickle Deserialization Attacks: Pickle's opcodes allow a pickle program to *import* and *invoke* arbitrary Python callables during unpickling. An attacker can use of pickle's *callable importing* and *callable invoking* opcodes to achieve arbitrary code execution—for example by executing an arbitrary shell command by invoking the `os.system` function. The dangers of deserializing arbitrary pickle programs have been publicized since 2011 [17, 30, 67, 75].

Protecting Pickle Models: Two existing approaches are used to protect users from malicious pickle models:

(1) **Model scanners** identify malicious models using denylists of unsafe callables. As with many denylist approaches, model scanners are useful for identifying recognizable malicious content, but are bypassed by malicious models that avoid denied callables, or that *indirectly* invoke callables [94].[3] Examples are Hugging Face's picklescan [41] and ProtectAI's scanner [73].
(2) **Restricted loaders** restrict the PM to execute only allowed, safe callables. The only available restricted pickle model loader is PyTorch's weights-only unpickler [79]. Its default allowlist is specialized for models produced by PyTorch.

3 Motivation

To summarize Section 2: it is dangerous to load untrusted pickle models, but alternative secure formats exist. Do pickle models remain a security threat? We answer this with a longitudinal study of pickle model usage (§3.1), and assessments of the usability of the PyTorch weights-only unpickler (§3.2),

3.1 Study of Pickle Models on Hugging Face

To determine whether pickle models are used despite the availability of alternate formats, we conducted a longitudinal study of the Hugging Face ecosystem. We investigated Hugging Face because it is the largest repository of pre-trained models [52], and because it hosts malicious pickle models [12, 16, 100, 104, 107]. At 10 points in time over a ~2-year period, we measured the download rates and model formats in repositories with ≥ 1000 monthly downloads (as a proxy for real-world impact).[4] The number of repositories in a measurement ranged from 2,296 in the first measurement to 16,661 in the last, with the number increasing monotonically at each point. We mined two existing datasets that covered January–October 2023

[3]We demonstrate this by creating two backdoored models that bypass two state-of-the-art scanners. One model uses callables that are missed by the scanners, and the other model uses disallowed callables by invoking them indirectly. See Appendix B of our extended report [54].
[4]Downloads are tracked using Hugging Face metrics.

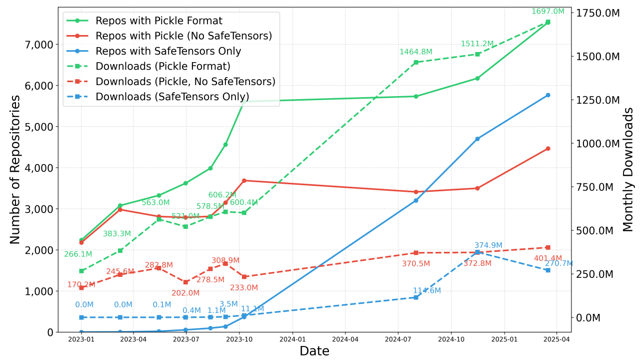


Figure 2: A longitudinal analysis of Hugging Face model formats for repositories with ≥ 1000 monthly downloads. Repositories can contain multiple models, each in different formats. Each color groups repositories by the model formats they contain: at least one pickle model (green), exclusively pickle (red), and exclusively SafeTensors (blue).

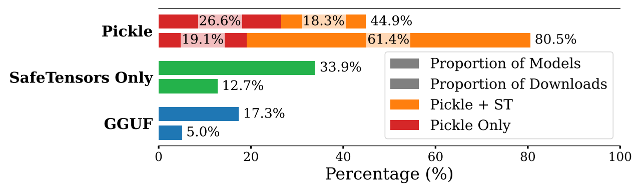


Figure 3: Proportions of model formats and downloads in March 2025, the final month of our longitudinal study. Notation: "Pickle+ST" indicates repositories with both formats.

(PTMTorrent [51] and HFCommunity [8]), and added new measurements in August 2024, November 2024, and March 2025 via the `huggingface_hub` API. In accordance with previous research [107], we determined model formats using file extensions; interested readers can refer to Appendix A of our extended report [54] for details.

Figure 2 summarizes our results. First, the red lines show that many important models continue to use only the pickle format, and these pickle-only models are downloaded 400M+ times per month. The green lines show that repositories containing both pickle and SafeTensors versions of models are also increasingly downloaded, with 1.70 billion monthly downloads. When models are converted to the SafeTensors format, the associated pickle model is often kept for legacy purposes and can still present security risks [61].

Figure 3 represents present-day usage with data from the final month of our study. Repositories with only SafeTensors or GGUF models are downloaded infrequently, in comparison to those with pickle models. Overall, ~44.9% of repositories contain pickle models, which aligns with previous estimates of 41%–55% [12, 107].

We anticipate that pickle models will continue to pose risks to the Hugging Face community for the next few years (cf. §7). Monthly download rates of pickle models are *increasing*, and many (21%) models are still exclusively in the pickle format, including 29 models in the top-100 most downloaded and over 500 models from Meta, Google, Microsoft, NVIDIA, and Intel. PyTorch remains the primary framework for model development, which reinforces reliance on pickle due to user familiarity [88]. Interoperability challenges persist during model conversion [20, 47, 101], complicating movement to other formats.

Summary: Despite positive steps to introduce secure alternative model formats like SafeTensors, pickle models are still prevalent and monthly downloads are *increasing*.

3.2 PyTorch Weights-Only Unpickler Usability

Our longitudinal study showed that pickle remains popular. Next, we assess whether the state-of-the-art safe loading approach, the PyTorch weights-only unpickler, can effectively load the pickle models we identify.

3.2.1 *Measurements.*

The weights-only unpickler, introduced in PyTorch 1.13 (Nov. 2022) and enabled by default in PyTorch 2.6 (Nov. 2024) [81], prevents access to dangerous callables, but can only load models that use callables from a small allowed set of PyTorch APIs.[5] Models that use additional callables cannot be loaded without user intervention; convenience and pressure from end-users results in library maintainers explicitly disabling the weights-only unpickler to maintain compatibility.[6] We investigate a sample of the pickle models in our study to determine whether they use callables disallowed by the weights-only unpickler, which affects the usability of the weights-only unpickler as a solution.

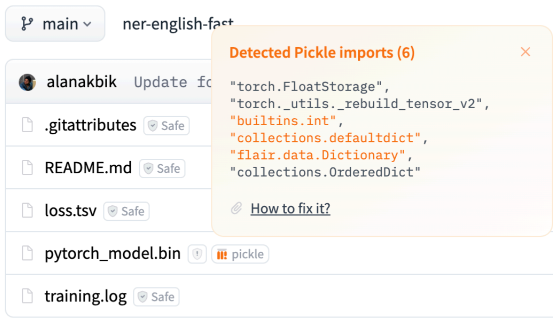
Methods: We sampled the most popular 1,500 of the 4,553 pickle-only repositories in our survey (§3.1). For each repository, we used the Hugging Face API to download its pickle models and used the fickling tool [67] to statically trace and inspect the callables used. We compared the callables in the model trace to the callables permitted by the weights-only unpickler's default policy. Models from 74 repositories failed to download or trace, leaving us with a sample of 1,426 repositories.

Results: Of the 1,426 model repositories surveyed, *219 repositories (15.4%) contained at least one pickle-based model that cannot be loaded by the weights-only unpickler* due to disallowed Python callables. These 219 repositories were downloaded 79.6 million times in the final month of our longitudinal study. In total, 36 unique disallowed callables appear in the 219 repositories. Many come from major libraries (e.g., numpy and Hugging Face Transformers). We list all disallowed callables that appear in traces in Figure 8 (Appendix A of our extended report [54]). These models cannot be loaded securely by the weights-only unpickler, so users must instead rely on the weaker model scanners (§2.2) and their own assessments of the models' safety.

3.2.2 *Motivating Example.*

We use an example to show the implications of these weights-only unpickler incompatibilities. Consider the `flair ner-english-fast` [33] model, a pre-trained pickle model for named entity recognition of English text that gained over 1 million downloads. To load the model, its documentation refers to the `flair` library's `SequenceTagger.load` API.

[5] PyTorch provides a mechanism for the user to manually expand the set of allowed callables [80], but the user is left to determine by themselves *which* callables to allow.
[6] As in the case of the `flairNLP` library (https://github.com/flairNLP/flair/commit/79aa33706e7f753f2edf962feb1d75de22af0d1d).



main ner-english-fast

alanakbik Update fc

.gitattributes Safe

README.md Safe

loss.tsv Safe

pytorch_model.bin pickle

training.log Safe

Detected Pickle imports (6)

```
"torch.FloatStorage",
"torch._utils._rebuild_tensor_v2",
"builtins.int",
"collections.defaultdict",
"flair.data.Dictionary",
"collections.OrderedDict"
```

How to fix it?

Figure 4: The Hugging Face repository for the `flair\ner-english-fast` model shows the results of the Hugging Face pickle scanning tool directly in the web application interface [33]. The pickle scanning tool warns that some imports in the model's pickle file are suspicious and require attention (highlighted).

The `flair` loading API exposes the user to risk. `Flair` models, like this one, use callables that are not part of the weights-only unpickler allowlist, so the API explicitly disables the weights-only unpickler to load its models. To protect themselves during loading, the user must depend on scanners. For this (benign) model, the Hugging Face pickle scanner [41] warns the user that some imports in the model pickle file are suspicious and require attention (Figure 4). We determined that the model is benign by manually reviewing its operations, and then reviewing the source code of the flair library to ensure that these operations are expected. This task is costly for every user to perform for every model. Our system successfully infers the expected operations and securely loads this model (§6.3) without manual effort.

Summary: While the weights-only unpickler offers security, 15.4% of sampled pickle repositories, with 79.6 million monthly downloads, contain a model that cannot use it. Library APIs disable the weights-only unpickler to load models, leaving users to rely on (incomplete) model scanners and manual assessments to determine if models are malicious.

4 System and Threat Model

Our motivational study shows the need for a new defense that is both usable and secure. Here, we model the system we aim to protect and the adversary to thwart:

System Model: The system loads a pickle-based model from an untrusted source (e.g., Hugging Face Model Hub) using APIs provided by a trusted ML library (e.g., PyTorch). We specifically aim to protect the system from the code introduced by the pickle program and executed by the Pickle Machine.

Threat Model: The attacker provides a maliciously crafted pickle to the victim with the intention of compromising the system. The attacker's goal is to execute arbitrary Python code (a "*payload*"), either directly during model loading, or after by e.g., overwriting a method in the model object with a reference to the payload.

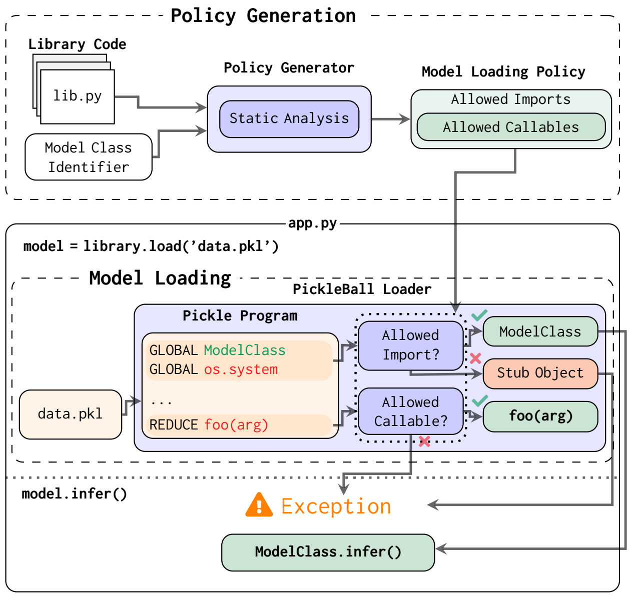


Figure 5: PickleBall works in two phases: 1) policy generation and 2) safe model loading. During policy generation, PickleBall takes as input the source code of a ML library and a class definition to analyze, and outputs a policy of allowed imports and invocations. During safe model loading, PickleBall enforces the extracted policy to protect the loading process. The loading application specifies the policy to enforce, based on the expected class of the model, and begins loading the model with the library API. The loading application can trust that any invocations of the Pickle Machine will be restricted to the configured policy.

- *In scope:* Manipulation of a pickle program in a pickle-based serialized model to execute arbitrary code.
- *Out of scope:* Manipulation of the data or code in the serialized model beyond the pickle program (e.g., model weights, data pipeline programs); manipulation of the trusted library code (e.g., PyTorch) used to load the serialized model.

We focus on the threat of pickle program code execution, and exclude other threats from untrusted ML models. Other ML supply chain attacks, like manipulating model weights to insert "backdoors" [15, 38], are orthogonal and can be approached with layered defenses. We do not consider other forms of attacks that manipulate pickle programs, e.g., for denial of service [18] due to their weaker attack primitives.

5 PickleBall Design and Implementation

PickleBall is designed to protect applications that use libraries to load untrusted pickle models. The desired system guarantee is that PickleBall raises a security exception when an adversary invokes an unnecessary callable during model loading, while transparently loading benign models. The idea behind PickleBall is to first generate a policy describing a minimal set of operations (i.e., the callables that need to be imported and invoked) for instantiating a given model object (§5.1), and then to enforce the generated policy during unpickling, rejecting spurious operations performed by malicious models (§5.2). Figure 5 provides an overview of the aforementioned PickleBall components. PickleBall guarantees that, given correct AST and type information, PickleBall raises a security exception when the adversary invokes an unnecessary callable (§5.3). We implement the design of PickleBall as a software artifact (§5.4).

5.1 Policy Generation

PickleBall's policy generation component is designed to automatically create a policy that describes the set of operations (i.e., the imported and invoked callables) permitted when loading a model of a particular library class. The policy is generated before loading the untrusted pickle model, and restricts the loading behaviors to only those that are necessary for the library API.

Design Rationale: We guide our design by studying how pickle models make use of Python callables. In regular usage, a pickle program needs a callable to construct non-primitive objects and (recursively) initialize the values of its attributes. Callables that are not needed to instantiate a given object should not appear in a pickle program; malicious payloads insert new code that does not have a role in object initialization.

The challenging task of policy generation is determining *which* callables are needed to instantiate an object. Python is a dynamic language that permits a single variable to receive different types at various program paths, and for objects to receive new attribute variables after initialization. Identifying all attribute types requires a path-sensitive analysis of the object creation code up to the point that the object is serialized. Further, in the ML setting, the code that creates the model object is not provided: the model saving program that trained the model is not published along with the model.

However, we recognize that we can generate approximate policies for serialized models by analyzing the class definition of the model in the library source code, even without access to the model saving program. The majority of the class instance attribute information is contained in the class definition, rather than in the model saving application. Intuitively, for the object to be reusable, it has to conform to an expected interface that is defined in the library.

We hypothesize that any unobserved object writes after the model is created will either not introduce new types to the variables, or will describe specialized metadata that is not necessary for the general-use operations that the model loading application is likely to perform, like inference. If new types are introduced, the model class interface shared between the saving and loading programs is violated. This is supported by our evaluation (§6.3.2).

Policy Generation Algorithm: Given an ML library and model class definition, PickleBall analyzes the class definition to generate a model loading policy as the sets of *allowed imports* and *allowed invocations*; these represent the operations that a pickle program needs to instantiate an instance of the class:

(1) **Allowed imports**: the set of callables permitted as arguments to Pickle Machine import operations.
(2) **Allowed invocations**: the set of callables permitted arguments to Pickle Machine invoking operations.

The set of *allowed invocations* is a sub-set of *allowed imports*—before being invoked, the callable must be imported. However, a callable that is imported but not invoked may only be used as a reference or as a constructor with its allocator method (`__new__`).

To statically generate the policy for a given class, PickleBall implements and applies rules starting at the class definition, and proceeds until the analysis terminates. PickleBall maintains a *candidate queue* of classes that is initialized with the class definition. PickleBall adds new classes to the queue as they are discovered by the analysis rules, and removes them as each class is analyzed. A class is only added to the candidate queue once to ensure that the analysis terminates.

PickleBall applies the following class-analysis rules:

(1) If the class implements a `__reduce__` method: identify the method return values (a callable, arguments for the callable, and optional state initialization values); add the returned callable to the *allowed imports* and *allowed invocations* sets; identify the types of all arguments for the callable and state initialization values, and add their class definitions to the *candidate queue*.
(2) Otherwise: add the class to the *allowed imports* set; add all sub-classes of the class to the *candidate queue*; add all types of the class's attributes (including attributes inherited) to the *candidate queue*.

The PickleBall policy generation algorithm, shown in Algorithm 1, operates over an abstract syntax tree (AST) representation of the analyzed program, and expects recovered type information to be labeled in the AST.

PickleBall's static analysis cannot produce perfectly sound and precise policies due to fundamental challenges in statically analyzing Python code, which is dynamically typed [9, 37, 102]. When recovered type information is over-approximate, PickleBall produces policies that contain more callables than is strictly necessary. Under-approximate type information may result from Python's dynamic features, like dynamic typing and runtime attribute manipulation, producing unaccounted data dependencies during type recovery. The missing type information creates policies that incorrectly exclude callables. To account for these potential errors, we design for security by separately enforcing allowed import and allowed invocations, and for robustness with a lazy enforcement mechanism (§5.2). We discuss how static analysis limitations affect the whole-system analysis in Section 5.3.

5.2 Policy Enforcement

PickleBall's policy enforcement module is designed to protect the model loading application during loading. It is a drop-in replacement for the system pickle module. It restricts the behavior of the PM operations that import, initialize, and invoke Python callables, so that only the callables permitted by the model loading policy are accessible to the pickle program in the model.

The module receives the loading policy and pickle program as inputs, and either outputs the deserialized object from the pickle program, or raises a security exception. "Importing opcodes" can only access callables that are allowed by the *allowed imports* policy. "Allocating opcodes" are allowed to create instances of objects listed in the *allowed imports* set by invoking their `__new__` methods.

Algorithm 1: The pseudocode algorithm designed to generate a model loading policy for a given class, performed over an AST augmented with recovered type information. Errors in the recovered type information introduce incorrectness in the results of `GetReduceReturnTypes` and `GetAttributeTypes`.

```
Input: ModelClass
Output: AllowedImports
Output: AllowedInvocations
Candidates := UniqueQueue(ModelClass)
AllowedImports := EmptySet
AllowedInvocations := EmptySet
while NotEmpty(Candidates) do
    Candidate := Pop(Candidates)
    if HasReduceMethods(Candidate) then
        AllowedImports+= GetReduceReturn(Candidate)
        AllowedInvocations+= GetReduceReturn(Candidate)
        Candidates+= GetReduceReturnTypes(Candidate)
    else
        AllowedImports+= Candidate
        Candidates+= GetSubclasses(Candidate)
        Candidates+= GetAttributeTypes(Candidate)
```

Callable-invoking opcodes are either removed entirely, or restricted to only invoke callables that are in the *allowed invocations*.

To enforce this, the module verifies that the name of a given callable is present in the allowed invocations set. Conceptually, an attacker could bypass the allowed imports/allowed invocations separation by importing a callable present in the allowed imports set but not in the allowed invocations set, then "renaming" it to a callable that is in the allowed invocations set. To mitigate against this, the policy enforcement module prevents building opcodes from modifying `__name__` and `__module__` attributes.

As described in Section 5.1, PickleBall policies may exclude valid callables when PickleBall fails to analyze dynamic Python features; PickleBall is designed to be robust against this with *lazy enforcement*. PickleBall aims to handle cases where the excluded callables are not accessed in the model's downstream use cases. When PickleBall's loader encounters a disallowed import operation, it creates a stub object instead of immediately raising an exception. The stub object records the name of the callable, and implements no functionality other than raising a security exception when invoked or accessed. This permits the loader to proceed to completion in the event that the stub object is never used, deferring the violation until access. The security exception is also raised when the stub object is accessed after initialization, preventing an attacker from overwriting methods of the returned object with denied callables that are later invoked.

5.3 Security Guarantees and Limitations

Whole-system Guarantees: PickleBall's design thwarts the adversary described in our Threat Model (§4) by restricting the PM's access to Python callables. The design guarantees that when provided a correct AST with type information for a Python class definition, PickleBall (1) raises a security exception when the adversary invokes a spurious callable, while (2) successfully instantiating any object that respects the attribute and type information defined class definition. In our evaluation of PickleBall, we show that existing state-of-the-art static analysis tools provide sufficient AST and type-recovery information for practical use (§6).

Policy Generation Guarantees: When PickleBall is provided a correct AST with type information, it is guaranteed to output a loading policy that includes all *allowed imports* and *allowed invocations* that can appear in when an object is saved in the pickle format, provided that the object is not manipulated to add attribute types outside of its object prototype.

Policy Enforcement Guarantees: When PickleBall loads a model, it is guaranteed to prevent the invocation of any Python callable that is not in the configured set of *allowed invocations*, and to create sanitized stub objects for any callable that is not in the configured set of *allowed imports*. The stub objects raise security exceptions when accessed/invoked.

Whole-system Limitations: PickleBall is limited fundamentally by the challenges of analyzing dynamic Python code with static analysis techniques, but PickleBall takes steps to mitigate these. Python's dynamic features, like runtime attribute manipulation and dynamic typing, prevent PickleBall's static analyses from creating an AST with sound and precise type information; this makes PickleBall's policies unsound and incomplete. Over-approximations in the AST result in policies that permit more Python callables than necessary; PickleBall mitigates this by having separate *allowed invocations* and *allowed imports* policies, so that only a small set of callables may be invoked. Under-approximations in the AST result in policies that omit benign callables from valid models; PickleBall mitigates this with *lazy policy enforcement* so that omitted callables only raise exceptions if they are invoked by the model, rather than just initialized but unused. We evaluate PickleBall in Section 6 to determine whether these limitations restrict it in practical settings (and find that they do not).

Remaining Attack Surface: PickleBall prevents attackers from importing and invoking arbitrary callables for malicious payloads. However, akin to return-to-`libc` attacks, PickleBall does not prevent the attacker from invoking permitted callables in sequences or with parameters that result in unintended outcomes. We are unaware of attacks leveraging these primitives, but whether this remaining capability is exploitable is a subject for further research (§7).

5.4 Implementation

PickleBall is implemented in a total of ~1,300 Scala lines of code (LoC) and ~300 Python LoC divided between two primary components: a static program analysis that builds upon the Joern framework [103]; and a dynamic loader that modifies the existing Pickle Machine. In the static analysis, ~700 Scala LoC implement Algorithm 1, ~600 Scala LoC extend and fix Joern features, and ~200 Python LoC integrate components. In the loader, ~100 Python LoC modify the Pickle Machine to implement lazy policy enforcement. Joern provides a program analysis platform for PickleBall by generating a Code Property Graph (CPG) with recovered type information for the target code; we extend Joern to improve type recovery features and class inheritance tracking.

(For more details, see Appendix C of our extended report [54]). Our analysis queries Joern's AST nodes for the relevant information.

Limitations: We inherit some limitations from the Joern program analysis framework. The limitations include that PickleBall:

- Cannot parse new Python syntax features (e.g., generic types).
- Cannot recognize type hints provided in docstring comments, but it can process type annotations introduced in Python 3.5.
- May fail to resolve dependencies, especially of builtin types.
- Cannot identify attributes of classes implemented in C.

These are engineering limitations and can be addressed with improvements to the underlying static analysis framework; they are not fundamental limitations of the PickleBall approach for determining model loading policies from library class definitions.

To account for these limitations when evaluating the PickleBall approach, we apply some manual library pre-processing before analysis (§6.1), and discuss future work to reduce the need for manual changes (§7).

6 Evaluation

We evaluate PickleBall with four Research Questions (RQs):

- **RQ1: Malicious Model Blocking.** How well does PickleBall block malicious pickled models from executing their payloads?
- **RQ2: Benign Model Loading.** How well does PickleBall correctly load benign pickled models?
- **RQ3: Performance.** Is PickleBall's runtime overhead practical for deployment?
- **RQ4: Comparison to SOTA.** How does PickleBall compare to the state-of-the-art security tools for protecting model loading applications?

6.1 Constructing an Evaluation Dataset

To answer these questions, we need a comprehensive dataset of pickle models consisting of both malicious and benign examples. The dataset must contain models created by different libraries to represent the diversity of loading APIs. We created our dataset by combining existing datasets, open models on Hugging Face, and constructing synthetic models. Our dataset contains 252 benign and 84 malicious models, for a total of 336.

Benign Models and Trusted Libraries: Our dataset must represent how users typically load and use models; therefore, we need a set of benign models and the libraries used to load and interact with them. We first searched for libraries that meet three criteria: (1) they load pickle models; (2) they have a model class type that is returned by a loading API; and (3) if a foundational library (e.g., PyTorch or transformers) type is used, the custom class adds new attributes to the type. These criteria are motivated by PickleBall's purpose: to restrict the pickle operations permitted by a library loading API based on analysis of the intended class type.

We identified candidate libraries by identifying popular pickle models on Hugging Face and working backwards. We first searched Hugging Face programmatically for pickle models, ordered by monthly download rate, that had model loading documentation directing users to a model loading library API. We manually reviewed the top 400 models (approximately 2 hours of review time) to determine whether the identified libraries meet our criteria; this resulted in 16 accepted libraries.

All libraries and their associated version information are listed in Table 3, Appendix D of our extended report [54]. Then, we identified candidate models associated with each library. We again queried Hugging Face to identify models associated with the library, either directly (as a piece of repository metadata) or by mention in the model documentation or name. We collected models with ≥ 100 monthly downloads at time of collection. In total, we accumulated 252 models produced by 16 different libraries. All collected models are listed in Table 4, Appendix D of our extended report [54].

We acknowledge that these models could themselves be malicious. We partially mitigate this by sampling from the most frequently downloaded models and libraries, checking model scanner indications, and manually investigating unexpected callables when restricted model loaders identify them.

Malicious Models: Our dataset must also represent the models created by our intended adversary (cf. our threat model — §4); therefore, we need a set of malicious pickle models. We first collected 82 malicious models and pickle programs that were identified on Hugging Face by two state-of-the-art model scanners [12, 107]. We add our 2 malicious models constructed to bypass scanners (see Appendix B of our extended report [54]), for a total of 84. All malicious models contain pickle programs with payloads that import and invoke Python spurious functions; payload behaviors include accessing sensitive files, making network connections, creating reverse shells, among others.

We acknowledge that these models do not represent the complete set of malicious behaviors; it is a best-effort collection of real-world pickle model malware to represent today's attackers. We aim to prevent adversaries that execute arbitrary Python functions during pickle loading, and the collected models all exercise this feature.

Test Harnesses: PickleBall protects programs that load untrusted models; therefore, we need a representative set of loading programs to secure. We create one test harness for each library in our dataset; the harness loads a model using the library API and performs an inference task.

Library Pre-processing: We pre-process the libraries before analyzing them to account for limitations in the static analysis framework (§5.4) and improve the correctness of the AST. We make manual source code modifications (<10 LoC) when the library class uses newer Python features of Python that Joern's front-end parser does not support, like generic type inheritance and type variables, or when the analysis misidentifies an imported library alias.

For libraries that provide type hints in docstrings, which Joern does not parse, we copy (but do not modify) the hints as type annotations (<100 LoC). We manually copy dependencies into the analysis scope when discovered during policy generation. Because model loading policies are compositional, we pre-compute policies for some frequently reused dependencies, including classes from the Python standard library and PyTorch, and save them in a class "cache" for PickleBall to access when it recognizes one of the classes in its analysis. Due to the complexity and prevalence of dynamically dispatched and C code implementations in PyTorch, we supplement our analysis of PyTorch modules with the weights-only unpickler policy in the class cache.

6.2 RQ1: Malicious Model Blocking

PickleBall must protect loading programs from pickle models with malicious payloads. To evaluate this, we assess whether any of the generated PickleBall policies allow malicious model executions.

6.2.1 Methods. For every harness program in our dataset, PickleBall generates a loading policy by analyzing the library and associated model class. We use PickleBall to enforce the generated policy while the harness attempts to load all malicious models in our dataset. For each malicious model, we consider the model blocked if PickleBall raises a security exception during model loading or inference, preventing the payload from executing.

Some library APIs only load the pickle model after validating that the accompanying model metadata is well-formatted (e.g., architecture, name, version). For these libraries, we directly invoke `pickle.loads` on the malicious model, while enforcing the associated PickleBall policy.

6.2.2 Results. For all generated policies, PickleBall *prevents all (100%) malicious models from executing their payloads.* Since PickleBall's generated policies do not contain the dangerous callables leveraged by the malicious models (e.g., `eval()`, `system()`), PickleBall's loader raises an exception for all malicious models.

RQ1 Summary: PickleBall generates policies that effectively prevent all malicious pickled models in our dataset from executing their payloads.

6.3 RQ2: Benign Model Loading

PickleBall must let users load and perform tasks with benign models. Its policies must not be so restrictive that the models are unusable. We therefore evaluate PickleBall's policies for loading and correctly using the benign models in our dataset (§6.3.1 and §6.3.2). To ensure robustness of the loaded model despite lazy enforcement, we further test the successfully loaded models that contain stub objects (§6.3.3 and §6.3.4).

6.3.1 Methods. We measure PickleBall's ability to generate and enforce policies that correctly load and execute benign models. For each library in our dataset, we generate a model loading policy. Then, we enforce the policy while using the library's test harness to load each library's models. Once loaded, we test the model by performing one inference task with a test input, and capture the output. For comparison, we then re-execute the test harness *without* enforcing any policy (by using the regular pickle module), and capture the inference result. We consider the model load a success when (1) PickleBall loads the model without raising exceptions, and (2) the inference results are equivalent between the PickleBall and unrestricted environments.

6.3.2 Results. PickleBall generates policies that, when enforced, correctly load and execute 79.8% of benign models in the dataset (Table 1). In most cases, the policies contain all callables (Table 1 – Imports and Invocations Allowed) that are seen in the model traces (Imports and Invocations Observed). In some cases, PickleBall's policies do not include callables that are included in the models (flair, PyAnnote, YOLOv5, and YOLOv11), resulting in the creation of a stub object that is occasionally invoked, hence leading to a security violation. We investigated the failed models/cases to determine their causes:

- **Attributes set after initialization**: PickleBall fails to identify attribute types that are set outside of the type declaration. For example, after initializing the model object, some libraries allow users to write training metadata to the model, including data for the optimizer used and paths to output files. In many cases (e.g., 12 flair models), PickleBall misses callable types set this way but still successfully loads the model, since this metadata is not used. However, for one flair model and three PyAnnote models, a metadata object is invoked during loading.
- **Follow-on pickle loading**: PickleBall fails to load two models from the MeloTTS library after they have been loaded, due to additional pickle loading during inference. PickleBall's policy includes all callables needed to load the MeloTTS models. However, during inference, an additional pickle model is loaded that invokes a disallowed callable, resulting in a security violation.
- **Library version drift**: one PyAnnote model fails to load for legacy reasons: it uses a callable that was included in models created with previous versions of PyAnnote. The callable's class declaration exists in the library code base as an unused stub, with a comment that it is needed for backward compatibility reasons, but is otherwise unused. Therefore, PickleBall's analysis failed to recognize it as necessary for model loading.
- **Namespace inconsistency**: the remaining `YOLOv5` and `YOLOv11` models use inconsistent naming conventions. For example, a policy includes the callable `yolov5.models.common.Conv`; however, the model refers to this callable as `models.common.Conv`, while referring to other callables by the full `yolov5.*` namespace.

6.3.3 Lazy Enforcement Robustness – Methods. Due to PickleBall's lazy enforcement, models can load successfully with incomplete attributes. To further ensure that the benign models are robustly instantiated with all attributes needed for inference, we evaluate these models with a more rigorous test suite of inputs. We investigated the libraries that successfully load models with stub objects, i.e., Flair, PyAnnote, YOLOv5, and YOLOv11.

For each library, we find an extensive evaluation dataset to test each loaded model: for Flair, we used various Named Entity Recognition and Universal Dependencies [28] datasets that come prepackaged with the Flair library [34]; for PyAnnote, we used the AISHELL-4 speech dataset [35]; for YOLOv5 and YOLOv11, we used the 2017 Test Images Common Objects in Context dataset [57].

For each model that PickleBall successfully loads, we evaluate the model on the dataset and ensure that the models do not raise security errors (i.e., they do not access any stub objects).

6.3.4 Lazy Enforcement Robustness – Results. All models yield the same results when loaded with PickleBall, compared to when loaded with the regular, unrestricted Pickle Machine. None of the models raise security errors during dataset evaluation, indicating that models are correctly instantiated, despite using stub objects.

RQ2 Summary: PickleBall generates policies that safely load and execute 79.8% of benign pickled models in our dataset.

Table 1: PickleBall generates loading policies for popular libraries (see GitHub Stars), which are evaluated by loading popular models (see cumulative model downloads in March 2025). We compare the number callables observed in the models to the callables allowed by the policies, and the number of stub objects that are created or invoked when the policies exclude callables. We compare PickleBall's loading success rate with the weights-only unpickler.

	Popularity		Imports			Invocations			Loading		
Library	Stars	Downloads	Observed	Allowed	Stub Objects	Observed	Allowed	Stub Calls	# Models	WOUp (%)	PickleBall (%)
CONCH [19]	342	13.7K	3	822	0	2	61	0	1	1 (100.0%)	1 (100.0%)
FlagEmbedding [32]	9.3K	11.1M	4	773	0	2	61	0	14	14 (100.0%)	14 (100.0%)
flair [66]	14.1K	3.05M	34	1186	17	6	62	2	18	0 (0.0%)	17 (94.4%)
GLiNER [105]	1.9K	760K	3	870	0	2	61	0	17	17 (100.0%)	17 (100.0%)
huggingsound [45]	447	56.1M	3	767	0	2	61	0	17	17 (100.0%)	17 (100.0%)
LanguageBind [55]	800	495K	4	992	0	2	61	0	8	8 (100.0%)	8 (100.0%)
MeloTTS [63]	5.9k	406K	3	852	0	2	61	0	8	8 (100.0%)	6 (75.0%)
Parrot_Paraphraser [22]	890	911K	3	774	0	2	61	0	1	1 (100.0%)	1 (100.0%)
PyAnnote [74]	7.2k	32.6M	18	1085	9	5	64	0	14	0 (0.0%)	10 (71.4%)
pysentimiento [82]	588	1.31M	4	777	0	2	61	0	4	4 (100.0%)	4 (100.0%)
sentence_transformers [84]	16.4k	204M	5	1087	0	2	61	0	76	76 (100.0%)	76 (100.0%)
super-image [31]	170	64.9K	3	1016	0	2	61	0	6	6 (100.0%)	6 (100.0%)
TNER [97]	387	25.0K	4	769	0	2	61	0	4	4 (100.0%)	4 (100.0%)
tweetnlp [11]	341	80.7K	4	778	0	2	61	0	1	1 (100.0%)	1 (100.0%)
YOLOv5 [95]	53.4k	24.8K	28	920	7	4	61	0	12	0 (0.0%)	4 (33.3%)
YOLOv11 (ultralytics) [96]	39.2k	38.4M	63	1816	13	11	61	6	51	0 (0.0%)	15 (29.4%)
Total									252	157 (62.3%)	201 (79.8%)
Average										75.0%	87.7%

6.4 RQ3: Performance

PickleBall must be fast enough for practical use in developer and user tasks. We analyze two aspects of PickleBall's performance: (1) the time to generate a policy for a class, which is an offline, one-time analysis cost (§5.1), and (2) the additional runtime overhead of enforcing a policy to load and use a model, compared to the regular Pickle Machine.

6.4.1 Methods. To measure the time to generate policies, we execute PickleBall's policy generator three times for each library in our dataset and compute the average between the three. We measure the real time using the Python `time` library. We run this experiment on a laptop with a 14-core Intel i7 CPU and 32GB of RAM, representing a commodity developer environment.

To measure the additional runtime overhead of PickleBall's policy enforcer, we isolate and record the time each harness program spends invoking the pickle `load` function during model loading. We first execute harness program with the unrestricted Pickle Machine environment to load a benign model. Then, we perform the same execution with PickleBall enabled. For fair comparison, we ensure that the unrestricted environment always uses the Python implementation of the Pickle Machine, instead of an optimized C implementation. We run this experiment on a server with a 32-core AMD EPYC 7502 processor and 256GB of RAM (Ubuntu 24.04); this is used for the attached hard-drive space for interacting with the hundreds of models in our dataset.

6.4.2 Results. PickleBall generates policies for all libraries in a median of 14.0 seconds, with minimum 9.0 seconds (CONCH library) and maximum 29.8 seconds (YOLOv11 library) (Figure 6). This policy generation execution time is reasonable for integration within project build systems, as the policy needs only to be generated when the analyzed library source code is modified. PickleBall's policy enforcer incurs negligible overhead, with a 0.42ms (1.75%) median runtime overhead compared to the unrestricted Pickle Machine, as depicted in Figure 7.

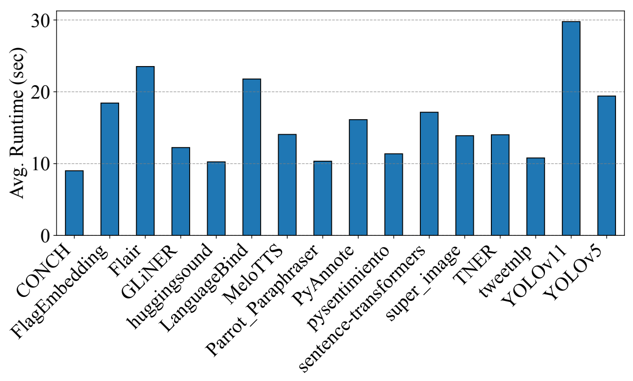


Figure 6: Time to generate a policy for each library class in dataset (averaged over 3 runs). This is a one-time step that can be integrated into existing workflows — either by library maintainers in the library's release process, or by a user, prior to loading the model.

RQ3 Summary: PickleBall policies are generated in a median 14.0 seconds across the evaluation libraries, PickleBall incurs a median runtime overhead of 0.42ms when loading models.

6.5 RQ4: Comparison to SOTA

We compare PickleBall with three existing state-of-the-art (SOTA) tools that share the same goal of defending against our threat model described in Section 4. As discussed in Section 2.2, existing pickle model defense tools fall into two categories: model scanners and restricted loading environments (like PickleBall).

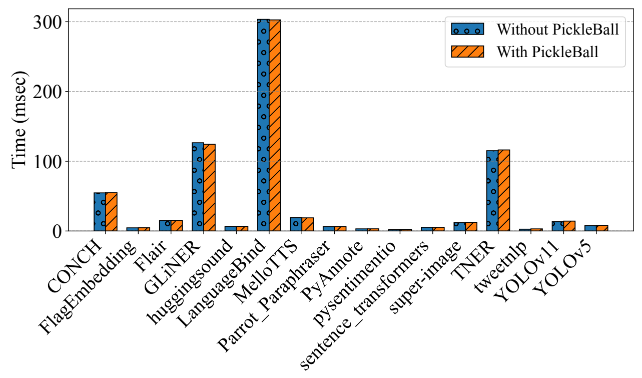


Figure 7: Time spent executing `pickle.load` in test loading program, with and without PickleBall (averaged over 3 runs after 1 warmup run). PickleBall incurs a median runtime overhead of 1.75% and average runtime overhead of 2.62%.

Table 2: Comparison of PickleBall to SOTA alternatives. Model scanning tools achieve low false positives on our dataset, but misclassify malicious models. Restricted loaders (including PickleBall) are secure, at the cost of blocking benign models. PickleBall loads more benign models than the weights-only unpickler due to its custom policies for each model class.

Tool	# TP	# TN	# FP	# FN	FPR	FNR
ModelScan [7]	75	236	16	9	6.3%	10.7%
ModelTracer [12]	44	252	0	40	0%	47.6%
Weights-Only Unpickler [79]	84	157	95	0	37.7%	0%
PickleBall (our work)	84	201	51	0	20.2%	0%

We compare against two model scanners: ModelScan [7], and the scanner implemented by Casey *et al.* [12] (henceforth ModelTracer).[7] ModelScan is a static analysis tool that applies a denylist to make determinations about models, and is integrated into Hugging Face. ModelTracer is a dynamic analysis tool that traces the model's invocations while it is loaded via `pickle.loads()/torch.load()`, and similarly applies a denylist.

We compare against one restricted loading environment: the weights-only unpickler [79]. The weights-only unpickler loads models by only permitting them access to callables in a rigid (but manually configurable) allowlist policy.

6.5.1 Methods. We evaluate the model scanning tools by providing each model in our dataset as an input to the tool. We expect the model scanners to alert when provided a malicious model input, and otherwise not to alert.

We consider correctly identified malicious models as true positives, correctly identified benign models as true negatives, incorrectly identified malicious models as false negatives, and incorrectly identified benign models as false positives.

[7]The authors provided access to the tool for evaluation purposes.

We evaluate the weights-only unpickler by attempting to load each model in our dataset using the PyTorch loading API with the weights-only unpickler enabled. We use the weights-only unpickler's default policy while loading models. We expect restricted loading environments like the weights-only unpickler and PickleBall to succeed when loading benign models and to raise exceptions when loading malicious models.

For parity when comparing with the model scanning tools, we consider raising an exception during malicious model loading as a true positive, correctly loading a benign model as a true negative, incorrectly rejecting a benign model as a false positive, and incorrectly loading a malicious model as a false negative.

6.5.2 Results. Comparisons of the tools are shown in Table 2. The model scanning tools resulted in few (16) false positives, while the restricted loaders resulted in 0 false negatives. Table 1 compares the success rate of PickleBall and the weights-only unpickler when loading benign models.

ModelScan incorrectly identified 9 malicious models as benign (false negatives) and did not report false positives. We identified three categories of ModelScan's false negatives: (1) five models implement payloads using callables that are not included in ModelScan's rigid denylist; (2) three models use dynamic runtime operations (e.g., `numpy.load()`) to load additional payloads that ModelScan fails to statically identify; and (3) one model uses multiple `STOP` pickle opcodes, resulting in ModelScan terminating its analysis after reaching the first one and missing the rest of the malicious payload. While this last model would not execute its malicious payload when executed by the Pickle Machine, it could be loaded in non-standard ways by another malicious pickle program to execute its payload. Categories (1) and (2) are fundamental limitations to using a static analysis denylist approach: the denylist cannot be complete and can be subverted.

ModelTracer successfully identified 44 malicious models but missed the remaining 40, resulting in a high false negative rate of 47.6%, but did not report false any positives. ModelTracer's false negatives appear from its limited denylist: it alerts on models that invoke the `execve`, `connect`, `socket`, or `chmod` system calls. ModelTracer does not consider file access operations to be indicators of malicious behavior, so malicious models that perform dangerous file reads and writes are not identified. This once again highlights the scanning limitation of relying on an incomplete denylist to indicate malicious behavior.

The weights-only unpickler, like PickleBall, prevents all malicious models from loading. However, it incorrectly blocks 95 benign models from loading, compared to PickleBall's 51. However, PickleBall's custom generated policies load additional models that the weights-only unpickler can not.

RQ4 Summary: While PickleBall prevents all malicious models from loading, model scanning tools fail to identify all malicious models. The weights-only unpickler is also effective at preventing malicious models from loading, but is less effective than PickleBall at loading benign models due to its limited default policy.

7 Discussion and Future Work

PickleBall's remaining attack surface: PickleBall reduces the available attack surface by significantly restricting access to callables, but it does not guarantee that the remaining callables cannot be composed in a malicious payload (§5.3). PickleBall could be compromised by code reuse techniques [21, 69] to stitch together permitted calls to construct an exploit. Thus far, no such attacks have been observed, but the question remains: can we generate malicious payloads that obey the policy constraints enforced by PickleBall and the weights-only unpickler?

Huang et al. [40] studied manual implementations of restricted unpicklers in the general (non-ML) setting, and devised attack strategies to overcome callable allowlists. When faced with well-implemented restricted unpicklers (i.e., when recursive attribute look-ups and indexing are disallowed by design, as in PickleBall and the weights-only unpickler), their approach degrades to manual policy inspection. Liu et al. [59] implemented PickleCloak to automatically detect useful pickle gadgets, but apply it to bypass scanner deny-lists rather than restricted loader allow-lists. Future work should identify properties of callables to automatically distinguish whether a callable can be used maliciously to bypass restricted allow-lists.

Long-term outlook for pickle in ML: Pickle remains a popular model format (§3), despite more secure alternatives. Each major model format provides tradeoffs in flexibility (pickle), security (SafeTensors), and efficiency (GGUF). Pickle is flexible, as it can serialize virtually any Python object, including complex models and non-standard data structures. SafeTensors was developed for security-sensitive deployments, with a structured, memory-mapped format. GGUF maximizes performance on inference-optimized runtimes. *Because these formats are complementary and used as defaults in different popular frameworks and ecosystems, we expect them to coexist going forward.* PickleBall does not discourage the adoption of secure alternatives to pickle, but provides a secure option for the large and growing pickle population.

One rapidly evolving area of ML models is large language models (LLMs), where pickle still appears despite major industry leaders releasing foundation models in the SafeTensors and GGUF formats. Popular foundation models like LLaMA-4 [62], Qwen-3 [83], and Deepseek-R1 [27] are encouragingly released with the SafeTensors format (although some, like LLaMA-3.1 [61], still provide a pickle model backup). However, foundation models are often adapted (e.g., fine-tuned) and redistributed, often with new formats and artifacts, including pickle. For example, we observed instances of models that are fine-tuned from LLama-4 [46, 60], Qwen-3 [85, 89], and Deepseek-R1 [93] and distributed with an additional pickle file that represents the training arguments used during fine-tuning. Even when secure formats are adopted for foundation LLM models, pickle continues to persist in the LLM ecosystem, which is consistent with our analysis (§3.1) and justifies the need for PickleBall.

Generalizing PickleBall: PickleBall is designed to protect pickle model loading and is evaluated on models found on Hugging Face, but its approach can generalize to protect (1) other model formats; and (2) other pickle applications. PickleBall aims to protect pickle deserialization for ML models, but its approach does not rely on any ML-specific properties.

This approach allows PickleBall to protect other applications that receive pickle data. To generalize, PickleBall's analysis requires that *the intended type of the pickle object is known before loading* (§5.1). In the ML setting, this is reasonable because the protected program is a *client application*. Other approaches are needed when the security analysis does not know *a priori* the intended type of the serialized object [24, 106].

PickleBall's approach works for the pickle format because it has (dangerously) expressive deserialization operations, and is used by trusted libraries that implement their own custom model classes. Other model formats that meet these criteria are candidates for protection in the PickleBall approach. Zhu et al. show that the TensorFlow SavedModel format has undesirable operations [108]; libraries that extend the TensorFlow model class with their own custom behaviors could use a PickleBall approach to restrict the allowed behaviors, but we are not aware of any that do. Formats like SafeTensors and GGUF are not known to have dangerous operations; if any were discovered, then a PickleBall approach might apply for identifying when to permit certain operations. We aim to explore which other model formats meet these criteria for PickleBall to assist in securely loading.

We evaluated PickleBall using models sourced from Hugging Face, but PickleBall will work similarly for models from any platform. The inputs to PickleBall are ML libraries and pickle models, which are ML artifacts that are not tied to the hosting platform. Hugging Face is the largest model hosting platform, with over 1.8M models available in July 2025. ModelScope [1] is a recent hub managed by China's Alibaba and hosts 80K models. It imitates Hugging Face's design and likewise has models with varying serialization formats. Other model communities, including Qualcomm AI Hub [4], PyTorch Hub [3], and TensorFlow Hub [5], have fewer than 500 models each and many are also hosted on Hugging Face. The ONNX Model Zoo [2] is now deprecated and archived on Hugging Face. Hugging Face models are representative of the kinds of models that PickleBall defends against.

Policy maintenance and distribution: PickleBall's intended workflow is that when a library is updated, its PickleBall policy would be updated as well. PickleBall makes policy maintenance easy for users with fast policy generation, incremental changes, and opportunities for seamless distribution.

PickleBall generates policies quickly, completing in under 30 seconds for each library in our evaluation (§6.4, Figure 6). This is reasonable for a task that must only occur when the library changes, and not every time PickleBall loads a model.

In practice, library updates result in either incremental policy changes or clearly documented breaks in supported model versions, leading to easier policy maintenance. After our evaluation concluded, we noticed one library, `FlagEmbedding`, receive updates (commit `bf6b649` to `875fd4f`), but PickleBall produced policies before and after with a 90% Jaccard similarity index, and which successfully loaded the same models in our dataset. When the library model class changes significantly, the library cannot load existing models, but we find it easy in practice to match models with a supported library version, due to the model documentation, as we do in our evaluation (§6.3.2). For example, we easily distinguish all models belonging to `YOLOv5` and its newer version, `ultralytics`.

PickleBall provides opportunities for easier policy maintenance if it gets adopted further in the model development life cycle. When library maintainers adopt PickleBall to generate policies automatically in the library release process, they can provide the updated policies alongside the libraries directly to users.

Removing library pre-processing: To account for implementation limitations (§5.4) when evaluating the fundamental PickleBall idea, we perform manual pre-processing of some libraries (§6.1), but future engineering work will remove this step. The purpose of the pre-processing is to overcome implementation limitations of the underlying static analysis framework that PickleBall depends on to produce an accurate type-annotated AST. To account for these limitations, we apply the following pre-processing steps:

- Remove generic type syntax from class inheritance statements.
- Copy type hints in comments into type annotation form.
- Copy relevant dependencies into the analysis scope.
- Reference the weights-only unpickler policy when a class inherits the `torch.nn.Module` class.

Manual source code modifications are applied to five out of 16 libraries in our dataset, and each account for between ∼10 and ∼100 modified LoC. Future engineering work to improve the underlying static analysis framework will remove the need for manual source code pre-processing.

8 Related Work

ML Model Loading Security: Pickle is not intended for untrusted data, but its proliferation as a model format created a security problem. To bring attention to the issue, security company Trail of Bits released the `fickling` tool for manipulating and analyzing pickle programs in ML models [67]. The fickling module provides manually-crafted allow-lists for select ML libraries, similarly to PickleBall, but requires expert-led audits to maintain the allow-lists, where PickleBall aims for automatic policy creating. New proposals for identifying malicious pickle models include dynamic [12] and static [107] scanners. Scanners take a model as input, and attempt to make an assessment of it based on fixed rules about malicious behaviors. Instead, PickleBall takes a model and the source code library that allegedly produced the model for context, and produces policies for the model based on that context. We showed in Section 6.5 that PickleBall's tailor-made policies result in no false negatives while comparative model scanners do produce false negatives due to their fixed rules.

Deserialization Attacks and Defenses: Deserialization vulnerabilities exist beyond the ML context. PainPickle [40] explored Python pickle security by creating a taxonomy of errors in custom Unpickler implementations, and devised attack strategies. We use their contributions (and suggestions) to guide the proper implementation of PickleBall's loader.

Other programming languages also have insecure deserialization APIs that need to be secured. Quack [24] proposes a generic deserialization defenses for PHP by employing a "static duck typing" static analysis, and Zhang et al. [106] propose a static analysis defense for Java. Python's pickle deserialization API is more expressive than those in PHP or Java, which are unable to directly invoke functions; this expressivity adds complexity to security policies.

Querying Code-graphs for Software Security: Graph representations are well-established for general program analysis tasks. For security specific tasks, Joern [103] introduced the Code Property Graph (CPG), a data structure that combines classic program analysis concepts into a representation that is easy queried to identify vulnerabilities. Follow on work ODGEN [56] extended Joern's CPGs into an Object Dependence Graph (ODG), capturing interactions from the object's point of view to detect vulnerabilities in Node.js packages. RogueOne [87] further evolved ODGs to form a data-flow relationship graph, fully capturing data-flows among objects. QL [64] and Datalog [86] based approaches inspired the CodeQL [13] query platform, which is used for vulnerability variant analysis tasks. PickleBall's implementation uses these code querying features and extends them to improve the accuracy of the program types recovered. Improvements to these program representations can lead to improved accuracy of PickleBall's policies.

9 Conclusion

Serialization and deserialization enable code and data exchange. Many recent works observe security vulnerabilities in deserialization, across various programming languages and contexts. We specifically examined the security vulnerabilities in pre-trained model deserialization that result from the use of (dynamically typed) Python and the reliance on Python's (insecure) pickle format. We found that pickle is common among the most popular models on Hugging Face, and that existing defenses are insecure or inapplicable to a substantial fraction of these models. Our PickleBall approach applies a novel program analysis to add greater security to model deserialization. In our evaluation, we demonstrated that PickleBall supported most existing benign models while preventing all known attacks in malicious models. We believe PickleBall is a promising complement to existing security resources in the pre-trained model ecosystem.[8]

Research Ethics

We considered the stakeholders involved in this study [25] and believe our results offer a net benefit. Our primary contribution is PickleBall, a defense that improves security for ML models. Our Hugging Face study abides by the platform policies for API use. We identified some shortcomings of existing defenses, and we followed a responsible disclosure process to report them.

Acknowledgements

We thank our anonymous reviewers for their valuable feedback, and Joseph Lucas, John Irwin, and Suha Hussain for their insightful discussions of pickle security. We thank Beatrice Casey for providing access to the ModelTracer tool [12], and to Shenao Wang for sharing malicious models discovered on Hugging Face [107]. This work is supported by the National Science Foundation (NSF), through award CNS-2238467. Andreas Kellas is supported by the Department of Defense (DoD) National Defense Science and Engineering (NDSEG) Fellowship program. Yaniv David is supported by the Center for Computer Engineering at the Technion. James Davis is supported by funding from Socket. Junfeng Yang is supported by funding from Google.

[8]We release all code and datasets at https://zenodo.org/records/16974645.

References

[1] 2025. ModelScope. https://www.modelscope.cn/home.

[2] 2025. ONNX Model Zoo. https://onnx.ai/models/.

[3] 2025. PyTorch Hub for Researchers. https://pytorch.org/hub/.

[4] 2025. Qualcomm AI Hub. https://aihub.qualcomm.com/models.

[5] 2025. TensorFlow Models and Datasets. https://www.tensorflow.org/resources/models-datasets.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[7] Protect AI. 2024. modelscan. https://github.com/protectai/modelscan. commit: 81338386b669526c14b839e7ccc36c160cd53b88.

[8] Adem Ait, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2023. Hfcommunity: A tool to analyze the hugging face hub community. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE.

[9] John Aycock. 2000. *International Python Conference* (2000).

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax.

[11] Jose Camacho-Collados, Kiamehr Rezaee, Talayeh Riahi, Asahi Ushio, Daniel Loureiro, Dimosthenis Antypas, Joanne Boisson, Luis Espinosa-Anke, Fangyu Liu, Eugenio Martínez-Cámara, et al. 2025. Python library tweetnlp provides a collection of useful tools to analyze/understand tweets such as sentiment analysis, etc. https://github.com/cardiffnlp/tweetnlp.

[12] Beatrice Casey, Joanna C. S. Santos, and Mehdi Mirakhorli. 2024. A Large-Scale Exploit Instrumentation Study of AI/ML Supply Chain Attacks in Hugging Face Models. arXiv:2410.04490 [cs.CR] https://arxiv.org/abs/2410.04490

[13] Walker Chabbott and James Fletcher. 2023. Multi-repository variant analysis: a powerful new way to perform security research across GitHub. https://github.blog/security/vulnerability-research/multi-repository-variant-analysis-a-powerful-new-way-to-perform-security-research-across-github/

[14] Kenneth Ward Church, Zeyu Chen, and Yanjun Ma. 2021. Emerging trends: A gentle introduction to fine-tuning. *Natural Language Engineering* 27, 6 (2021).

[15] Eleanor Clifford, Ilia Shumailov, Yiren Zhao, Ross Anderson, and Robert Mullins. 2024. ImpNet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. 344–357.

[16] David Cohen. 2024. Data Scientists Targeted by Malicious Hugging Face ML Models with Silent Backdoor. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/

[17] ColdwaterQ. 2022. BACKDOORING Pickles: A decade only made things worse. https://forum.defcon.org/node/241825.

[18] coldwaterq. 2024. GitHub Issue: Fickling DoS. https://github.com/trailofbits/fickling/issues/111.

[19] Conch. 2025. A Vision-Language Foundation Model for Computational Pathology. https://github.com/mahmoodlab/CONCH/.

[20] Fredrik Dahlgren, Suha Hussain, Heidy Khlaaf, and Evan Sultanik. 2023. *EleutherAI, Hugging Face Safetensors Library*. Technical Report. Trail of Bits.

[21] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code reuse attacks in php: Automated pop chain generation. In *2014 ACM SIGSAC Conference on Computer and Communications Security*. 42–53.

[22] Prithiviraj Damodaran. 2025. Parrot: Paraphrase generation for NLU. https://github.com/PrithivirajDamodaran/Parrot_Paraphraser/.

[23] Dark Reading Staff. 2024. Sleepy Pickle: Exploit Subtly Poisons ML Models. https://www.darkreading.com/threat-intelligence/sleepy-pickle-exploit-subtly-poisons-ml-models

[24] Yaniv David, Neophytos Christou, Andreas D. Kellas, Vasileios P. Kemerlis, and Junfeng Yang. 2024. QUACK: Hindering Deserialization Attacks via Static Duck Typing. In *2024 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA.

[25] James C Davis, Sophie Chen, Huiyun Peng, Paschal C Amusuo, and Kelechi G Kalu. 2025. A Guide to Stakeholder Analysis for Cybersecurity Researchers. *[arXiv'25] arXiv preprint arXiv:2508.14796v1* (2025).

[26] James C Davis, Purvish Jajal, Wenxin Jiang, Taylor R Schorlemmer, Nicholas Synovic, and George K Thiruvathukal. 2023. Reusing deep learning models: Challenges and directions in software engineering. In *2023 IEEE John Vincent Atanasoff International Symposium on Modern Computing (JVA)*. IEEE, 17–30.

[27] deepseek ai. 2025. DeepSeek-R1-Distill-Qwen-32B. https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B. commit: 711ad2ea6aa40cfca18895e8aca02ab92df1a746.

[28] Universal Dependencies. 2025. Universal Dependencies. https://universaldependencies.org/.

[29] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[30] Nelson Elhage. 2023. What's with ML software and pickles? https://blog.nelhage.com/post/pickles-and-ml/.

[31] eugenesiow et al. 2025. Image super resolution models for PyTorch. https://github.com/eugenesiow/super-image/.

[32] FlagEmbedding. 2025. BGE: One-Stop Retrieval Toolkit For Search and RAG. https://github.com/FlagOpen/FlagEmbedding.

[33] Flair. 2024. ner-english-fast. https://huggingface.co/flair/ner-english-fast. commit: f75577be7dbb6f47ea7681664560349e870aef18.

[34] Flair. 2025. How to load a prepared dataset. https://flairnlp.github.io/docs/tutorial-training/how-to-load-prepared-dataset.

[35] Yihui Fu, Luyao Cheng, Shubo Lv, Yukai Jv, Yuxiang Kong, Zhuo Chen, Yanxin Hu, Lei Xie, Jian Wu, Hui Bu, Xin Xu, Jun Du, and Jingdong Chen. 2021. AISHELL-4: An Open Source Dataset for Speech Enhancement, Separation, Recognition and Speaker Diarization in Conference Scenario. *CoRR* abs/2104.03603 (2021). arXiv:2104.03603 https://arxiv.org/abs/2104.03603

[36] GGML. 2025. GGUF. https://github.com/ggml-org/ggml/blob/master/docs/gguf.md.

[37] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. 2010. Alias analysis for optimization of dynamic languages. In *6th Symposium on Dynamic Languages*. Association for Computing Machinery, New York, NY, USA, 27–42.

[38] Tianyu Gu, Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2019. Badnets: Evaluating backdooring attacks on deep neural networks. *IEEE Access* 7 (2019).

[39] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[40] Nan-Jung Huang, Chih-Jen Huang, and Shih-Kun Huang. 2022. Pain Pickle: Bypassing Python Restricted Unpickler for Automatic Exploit Generation. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. 1079–1090.

[41] Hugging Face. 2024. Pickle Scanning. https://huggingface.co/docs/hub/en/security-pickle.

[42] Hugging Face. 2024. Safetensors. https://huggingface.co/docs/safetensors.

[43] Hugging Face. 2025. GGUF. https://huggingface.co/docs/hub/en/gguf.

[44] Hugging Face. 2025. Hugging Face Hub documentation. https://huggingface.co/docs/hub/en/index.

[45] HuggingSound. 2025. HuggingSound: A toolkit for speech-related tasks based on Hugging Face's tools. https://github.com/jonatasgrosman/huggingsound.

[46] Ilya-bs1. 2025. llama4-scout-interpreter-model. https://huggingface.co/Ilya-bs1/llama4-scout-interpreter-model. commit: d61cd8c577fcce5910f990718346034d710932c1.

[47] Purvish Jajal, Wenxin Jiang, Arav Tewari, Erik Kocinare, Joseph Woo, Anusha Sarraf, Yung-Hsiang Lu, George K Thiruvathukal, and James C Davis. 2024. Interoperability in Deep Learning: A User Survey and Failure Analysis of ONNX Model Converters. In *33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1466–1478.

[48] Wenxin Jiang, Berk Çakar, Mikola Lysenko, and James C Davis. 2025. Detecting Active and Stealthy Typosquatting Threats in Package Registries. *arXiv preprint arXiv:2502.20528* (2025).

[49] Wenxin Jiang, Chingwo Cheung, Mingyu Kim, Heesoo Kim, George K Thiruvathukal, and James C Davis. 2024. Naming Practices of Pre-Trained Models in Hugging Face. *arXiv preprint arXiv:2310.01642* (2024).

[50] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K Thiruvathukal, and James C Davis. 2023. An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2463–2475.

[51] Wenxin Jiang, Nicholas Synovic, Purvish Jajal, Taylor R Schorlemmer, Arav Tewari, Bhavesh Pareek, George K Thiruvathukal, and James C Davis. 2023. PTMTorrent: a dataset for mining open-source pre-trained model packages. In *IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*.

[52] Wenxin Jiang, Nicholas Synovic, Rohan Sethi, Aryan Indarapu, Matt Hyatt, Taylor R Schorlemmer, George K Thiruvathukal, and James C Davis. 2022. An empirical study of artifacts and security risks in the pre-trained model supply chain. In *2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 105–114.

[53] Kaitai Project. 2025. Python pickle serialization format: format specification. http://formats.kaitai.io/python_pickle/.

[54] Andreas D. Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P. Kemerlis, James C. Davis, and Junfeng Yang. 2025. PickleBall: Secure Deserialization of Pickle-based Machine Learning Models (Extended Report). (2025). arXiv:2508.15987 [cs.CR] https://arxiv.org/

abs/2508.15987
[55] LanguageBind. 2025. Extending Video-Language Pretraining to N-modality by Language-based Semantic Alignment. https://github.com/PKU-YuanGroup/LanguageBind/.
[56] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song. In *31st USENIX Security Symposium (SEC '22)*. USENIX Association, Boston, MA, 143–160.
[57] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. *CoRR* abs/1405.0312 (2014). arXiv:1405.0312 http://arxiv.org/abs/1405.0312
[58] Guannan Liu, Xing Gao, Haining Wang, and Kun Sun. 2022. Exploring the Unchartered Space of Container Registry Typosquatting. https://www.usenix.org/conference/usenixsecurity22/presentation/liu-guannan. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 35–51.
[59] Tong Liu, Guozhu Meng, Peng Zhou, Zizhuang Deng, Shuaiyin Yao, and Kai Chen. 2025. The Art of Hide and Seek: Making Pickle-Based Model Supply Chain Poisoning Stealthy Again. arXiv:2508.19774 [cs.CR] https://arxiv.org/abs/2508.19774
[60] madilcy. 2025. arabic-medical-llama4. https://huggingface.co/madilcy/arabic-medical-llama4. commit: 826ac2b97a5724aba87ceeb9001aebeb1300b7d5.
[61] meta llama. 2024. Llama-3.1-8B-Instruct - original/. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct/tree/main/original. commit: 0e9e39f249a16976918f6564b8830bc894c89659.
[62] meta llama. 2025. Llama-4-Scout-17B-16E-Instruct. https://huggingface.co/meta-llama/Llama-4-Scout-17B-16E-Instruct. commit: 92f3b1597a195b523d8d9e5700e57e4fbb8f20d3.
[63] MIT and Myshell.ai. 2025. MeloTTS is a high-quality multi-lingual text-to-speech library. https://github.com/myshell-ai/MeloTTS/.
[64] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. 2007. Keynote Address: .QL for Source Code Analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. 3–16.
[65] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. 2023. Beyond Typosquatting: An In-depth Look at Package Confusion. https://www.usenix.org/conference/usenixsecurity23/presentation/neupane. In *32nd USENIX Security Symposium (USENIX Security)*.
[66] Humboldt University of Berlin and friends. 2025. A very simple framework for state-of-the-art NLP. https://github.com/flairNLP/flair.
[67] Trail of Bits. 2024. fickling. https://github.com/trailofbits/fickling.
[68] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020*. Springer.
[69] Sunnyeo Park, Daejun Kim, Suman Jana, and Sooel Son. 2022. FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 197–214.
[70] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[71] Patry, Nicolas and Biderman, Stella. 2023. Audit shows that safetensors is safe and ready to become the default. https://huggingface.co/blog/safetensors-security-audit.
[72] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. 2021. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350* (2021).
[73] Protect AI. 2024. Protect AI and Hugging Face: Securing the ML Supply Chain. https://protectai.com/blog/protect-ai-hugging-face-ml-supply-chain.
[74] pyannoteAI. 2025. pyannote.audio speaker diarization toolkit. https://github.com/pyannote/pyannote-audio.
[75] Python. 2025. pickle — Python object serialization. https://docs.python.org/3/library/pickle.html.
[76] Python. 2025. Python Glossary – Callable. https://docs.python.org/3/glossary.html#term-callable.
[77] PyTorch. 2024. PyTorch serialization.py. https://github.com/pytorch/pytorch/blob/726424f4deac82b7cd74cc86a55c610085698535/torch/serialization.py#L6.
[78] PyTorch. 2024. TorchScript. https://pytorch.org/docs/stable/jit.html.
[79] PyTorch. 2024. Weights-only Unpickler. https://github.com/pytorch/pytorch/blob/main/torch/_weights_only_unpickler.py.
[80] PyTorch. 2025. PyTorch Commit 66dc8fb. https://github.com/pytorch/pytorch/commit/66dc8fb7ff822033c4b161fc216e21d6886568c7.
[81] PyTorch. 2025. Serialization semantics. https://github.com/pytorch/pytorch/blob/eb2df46b6af691cc13abfc8435c33963b30c7cb1/docs/source/notes/serialization.rst#torchload-with-weights_onlytrue.
[82] Juan Manuel Pérez, Mariela Rajngewerc, Juan Carlos Giudici, Damián A. Furman, Franco Luque, Laura Alonso Alemany, and María Vanina Martínez. 2025. pysentimiento: A Python toolkit for Sentiment Analysis and Social NLP tasks. https://github.com/pysentimiento/pysentimiento.
[83] Qwen. 2025. Qwen3-0.6B. https://huggingface.co/Qwen/Qwen3-0.6B. commit: e6de91484c29aa9480d55605af694f39b081c455.
[84] Nils Reimers and Iryna Gurevych. 2025. Sentence Transformers: Embeddings, Retrieval, and Reranking. https://github.com/UKPLab/sentence-transformers/.
[85] sarahbadr. 2025. MNLP_M2_dpo_model. https://huggingface.co/sarahbadr/MNLP_M2_dpo_model. commit: 1d67b4e322ebe2613c45a950a71ef204f93d1562.
[86] Max Schäfer and Oege de Moor. 2010. Type inference for datalog with complex type hierarchies. *SIGPLAN Not.* 45, 1 (Jan. 2010), 145–156.
[87] Raphael J. Sofaer, Yaniv David, Mingqing Kang, Jianjia Yu, Yinzhi Cao, Junfeng Yang, and Jason Nieh. 2024. RogueOne: Detecting Rogue Updates via Differential Data-flow Analysis Using Trust Domains. In *46th IEEE/ACM International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13.
[88] Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. 2022. An exploratory study of deep learning supply chain. In *44th International Conference on Software Engineering*. 86–98.
[89] TarhanE. 2025. sft-base_loss-Qwen3-0.6B. https://huggingface.co/TarhanE/sft-base_loss-Qwen3-0.6B-mle0-ul0-tox0-e4. commit: b2f51e83b726679bd64c9d34f3775e3d95b58a66.
[90] The HDF Group. 2025. HDF5. https://www.hdfgroup.org/solutions/hdf5/.
[91] Trail of Bits. 2023. EleutherAI, Hugging Face Safetensors Library Security Assessment. https://github.com/trailofbits/publications/blob/master/reviews/2023-03-eleutherai-huggingface-safetensors-securityreview.pdf.
[92] Adelin Travers. 2021. ONNX runtime hacks. https://github.com/alkaet/LobotoMl/tree/main/ONNX_runtime_hacks
[93] tttx. 2025. models-p10-ttt-18feb-fixed-sft-clip-step1. https://huggingface.co/tttx/models-p10-ttt-18feb-fixed-sft-clip-step1. commit: 0c3f92ef17faac10ad927bde668144f37cac0040.
[94] Dor Tumarkin. 2024. "Free Hugs" – What to be Wary of in Hugging Face – Part 4. https://checkmarx.com/blog/free-hugs-what-to-be-wary-of-in-hugging-face-part-4/
[95] Ultralytics. 2025. Real-time object detection and image segmentation model. https://github.com/ultralytics/yolov5.
[96] Ultralytics. 2025. (SOTA) Real-time object detection and image segmentation model. https://github.com/ultralytics/ultralytics.
[97] Asahi Ushio and Jose Camacho-Collados. 2025. T-NER: An All-Round Python Library for Transformer-based Named Entity Recognition. https://github.com/asahi417/tner/.
[98] Haifeng Wang, Jiwei Li, Hua Wu, Eduard Hovy, and Yu Sun. 2022. Pre-trained language models and their applications. *Engineering* (2022).
[99] Zhi Wang, Chaoge Liu, Xiang Cui, Jie Yin, and Xutong Wang. 2022. Evilmodel 2.0: bringing neural network models into malware attacks. *Computers & Security* 120 (2022), 102807.
[100] Eoin Wickens and Tom Bonner. 2024. Machine Learning Threat Roundup: February 2023: reverse shells and a steganography payload discovered in-the-wild. https://hiddenlayer.com/research/machine-learning-threat-roundup/
[101] Eoin Wickens and Kasimir Schulz. 2024. Hijacking SafeTensors Conversion on Hugging Face. https://hiddenlayer.com/research/silent-sabotage/
[102] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. Python probabilistic type inference with natural language support. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.
[103] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604.
[104] Karlo Zanki. 2025. Malicious ML models discovered on Hugging Face platform. https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face
[105] Urchade Zaratiana, Nadi Tomeh, Pierre Holat, and Thierry Charnois. 2025. Generalist and Lightweight Model for Named Entity Recognition. https://github.com/urchade/GLiNER/.
[106] Quan Zhang, Yiwen Xu, Zijing Yin, Chijin Zhou, and Yu Jiang. 2024. Automatic Policy Synthesis and Enforcement for Protecting Untrusted Deserialization. In *2024 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA.
[107] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. 2024. Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs. In *IEEE/ACM International Conference on Automated Software Engineering*.
[108] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Xiaofei Xie, and Rui Chang. 2025. My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs. In *2025 IEEE Symposium on Security and Privacy (SP)*.