# PICKLEBALL: Secure Deserialization of Pickle-based Machine Learning Models

**Andreas D. Kellas**[1], Neophytos Christou[2], Wenxin Jiang[3], Penghui Li[1], Laurent Simon[4], Yaniv David[5], Vasileios P. Kemerlis[2], James C. Davis[3], Junfeng Yang[1]

1 2 BROWN 3 PURDUE UNIVERSITY 4 Google 5 TECHNION Israel Institute of Technology

COLUMBIA ENGINEERING



Intro:

I'm Andreas, a 5th year PhD student at Columbia University.

Today I am presenting PickleBall, a novel defense against pickle-based model backdoors.

This work was collaborative between teams at Columbia, Brown, Purdue, Google, and Technion.

## Contributions

1. **Study** of model serialization formats in the Hugging Face model ecosystem.

2. **Secure pickle model loading** with library-specific policies.

3. **Dataset** of >300 malicious and benign pickle-based models and programs.



In this work, we contribute:

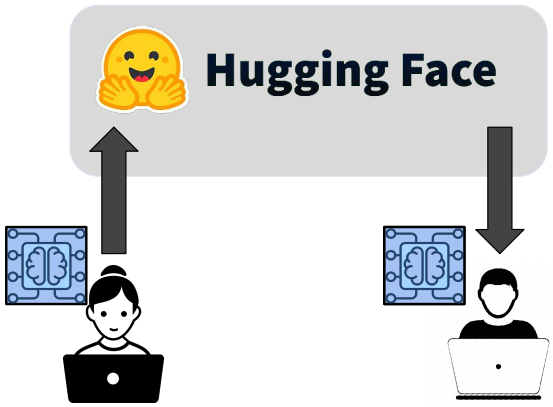
A study of the open model ecosystem on Hugging Face, and of the model serialization formats that are used.

A system for securely loading pickle models, by generating policies specific to libraries, and enforcing the policies

A dataset of over 300 malicious and benign pickle-based models and pickle programs to evaluate solutions.

## The Model Ecosystem Facilitates Model Sharing

The open model ecosystem lets engineers share and reuse pre-trained models.






Today, training models is expensive. The open model ecosystem lets us save money and time by letting us share and reuse pre-trained models from large model hubs, like Hugging Face.

However, models from these hubs may come from untrusted sources. This lets attackers abuse this trust by creating malicious backdoored models and coercing users into loading them.

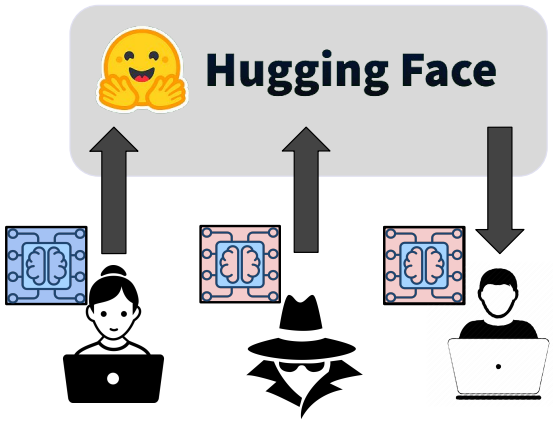
Backdoored models might be weights based, which compromise the inference of a model, or they can compromise the system that loads the models by executing arbitrary code.

This work focuses on backdoors that compromise the system.

## The Model Ecosystem Facilitates Model Sharing

The open model ecosystem lets engineers share and reuse pre-trained models.

Models may come from ***untrusted sources***, inviting security risks.






Today, training models is expensive. The open model ecosystem lets us save money and time by letting us share and reuse pre-trained models from large model hubs, like Hugging Face.

However, models from these hubs may come from untrusted sources. This lets attackers abuse this trust by creating malicious backdoored models and coercing users into loading them.

Backdoored models might be weights based, which compromise the inference of a model, or they can compromise the system that loads the models by executing arbitrary code.

This work focuses on backdoors that compromise the system.

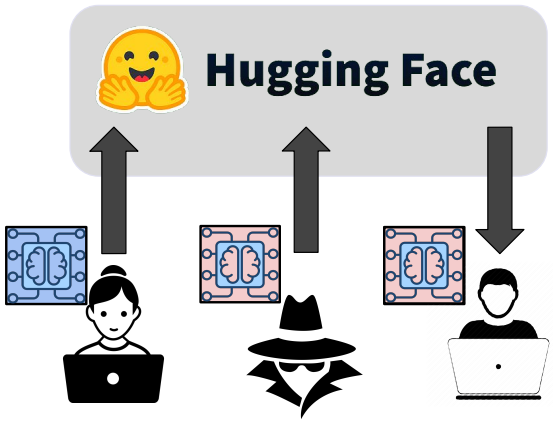
## The Model Ecosystem Facilitates Model Sharing

The open model ecosystem lets engineers share and reuse pre-trained models.

Models may come from ***untrusted sources***, inviting security risks.

Weights backdoors

System compromise backdoors






Today, training models is expensive. The open model ecosystem lets us save money and time by letting us share and reuse pre-trained models from large model hubs, like Hugging Face.

However, models from these hubs may come from untrusted sources. This lets attackers abuse this trust by creating malicious backdoored models and coercing users into loading them.

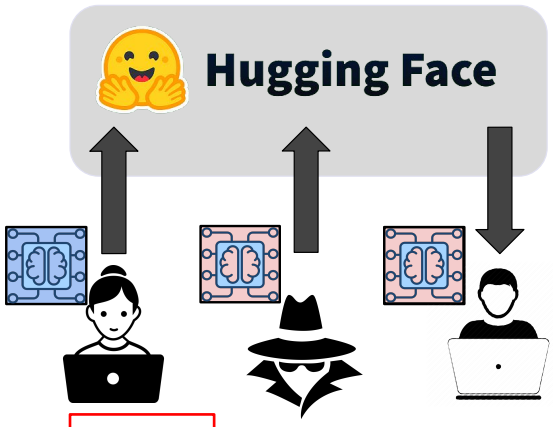
Backdoored models might be weights based, which compromise the inference of a model, or they can execute arbitrary code to compromise the system that loads the models.

This work focuses on backdoors that compromise the system.

## The Model Ecosystem Facilitates Model Sharing

The open model ecosystem lets engineers share and reuse pre-trained models.

Models may come from ***untrusted sources***, inviting security risks.




Weights backdoors

System compromise backdoors

**Focus of this work**



Today, training models is expensive. The open model ecosystem lets us save money and time by letting us share and reuse pre-trained models from large model hubs, like Hugging Face.

However, models from these hubs may come from untrusted sources. This lets attackers abuse this trust by creating malicious backdoored models and coercing users into loading them.

Backdoored models might be weights based, which compromise the inference of a model, or they can execute arbitrary code to compromise the system that loads the models.

This work focuses on backdoors that compromise the system.

## Attackers Abuse Pickle-Based Models

Models are saved and loaded with ***serialization formats***.



Attackers abuse the model loading process, where a saved model is deserialized from a file into the program memory.

While many model serialization formats exist, like SafeTensors, GGUF, and others, the most widely abused is pickle. Almost all discovered backdoors abuse the pickle format.

## Attackers Abuse Pickle-Based Models

Models are saved and loaded with ***serialization formats***.

Attackers abuse the **pickle** format to create backdoored models.

Threat Research | February 6, 2025

**Malicious ML models discovered on Hugging Face platform**

Software development teams working on machine learning take note: RL threat researchers have identified nullifAI, a novel attack technique used on Hugging Face.

BLOG AUTHOR
Karlo Zanki, Reverse Engineer at ReversingLabs

**Data Scientists Targeted by Malicious Hugging Face ML Models with Silent Backdoor**

SHARE:

By David Cohen, JFrog Senior Security Researcher | February 27, 2024

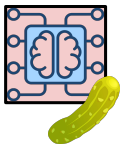


Attackers abuse the model loading process, where a saved model is deserialized from a file into the program memory.

While many model serialization formats exist, like SafeTensors, GGUF, and others, the most widely abused is pickle. Almost all discovered backdoors abuse the pickle format.

## Pickle is *Easy* to Abuse

The pickle format is (too) expressive:





So, why do attackers abuse pickle?
- Pickle is a very expressive format. Where other serialization formats may only encode the data in model weights and some information about operations, pickle models mix executable code and data.

Models that use the pickle format are encoded into a series of opcodes. When the model is loaded, these opcodes are executed by a virtual machine implemented in Python's pickle module - called the "Pickle Machine". When these opcodes execute, they instantiate the model as a python object and return it to the python program.

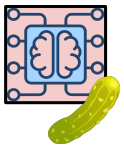
Most of the opcodes are very simple, but they expose functionality to import and invoke Python functions and class constructors (named "callables"). These are needed to construct complex data types like models, and load the weights values from disk.

These importing and invoking opcodes also give attackers an easy primitive to execute malicious callables – for example, by importing and calling functions like os.system.

## Pickle is *Easy* to Abuse

The pickle format is (too) expressive:

- Pickled objects are sequences of **opcodes** executed by the "Pickle Machine".





So, why do attackers abuse pickle?
- Pickle is a very expressive format. Where other serialization formats may only encode the data in model weights and some information about operations, pickle models mix executable code and data.

Models that use the pickle format are encoded into a series of opcodes. When the model is loaded, these opcodes are executed by a virtual machine implemented in Python's pickle module - called the "Pickle Machine". When these opcodes execute, they instantiate the model as a python object and return it to the python program.

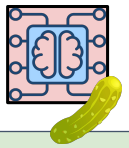
Most of the opcodes are very simple, but they expose functionality to import and invoke Python functions and class constructors (named "callables"). These are needed to construct complex data types like models, and load the weights values from disk.

These importing and invoking opcodes also give attackers an easy primitive to execute malicious callables – for example, by importing and calling functions like os.system.

## Pickle is *Easy* to Abuse

The pickle format is (too) expressive:

- Pickled objects are sequences of **opcodes** executed by the "Pickle Machine".
- Opcodes can **import** and **invoke** Python *callables* (functions or classes).



```
IMPORT torch.FloatStorage
CREATE torch.FloatStorage.__new__

IMPORT torch.rebuild_tensor_v2
INVOKE torch.rebuild_tensor_v2(...)
```



So, why do attackers abuse pickle?
- Pickle is a very expressive format. Where other serialization formats may only encode the data in model weights and some information about operations, pickle models mix executable code and data.

Models that use the pickle format are encoded into a series of opcodes. When the model is loaded, these opcodes are executed by a virtual machine implemented in Python's pickle module - called the "Pickle Machine". When these opcodes execute, they instantiate the model as a python object and return it to the python program.

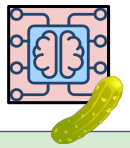
Most of the opcodes are very simple, but they expose functionality to import and invoke Python functions and class constructors (named "callables"). These are needed to construct complex data types like models, and load the weights values from disk.

These importing and invoking opcodes also give attackers an easy primitive to execute malicious callables – for example, by importing and calling functions like os.system.

## Pickle is *Easy* to Abuse

The pickle format is (too) expressive:

- Pickled objects are sequences of **opcodes** executed by the "Pickle Machine".
- Opcodes can **import** and **invoke** Python *callables* (functions or classes).
- Attackers can trivially import and invoke malicious callables, like `os.system`.



```
IMPORT torch.FloatStorage
CREATE torch.FloatStorage.__new__

IMPORT torch.rebuild_tensor_v2
INVOKE torch.rebuild_tensor_v2(...)

IMPORT os.system
INVOKE os.system('nc -lp 1337 -e /bin/bash')
```



So, why do attackers abuse pickle?
- Pickle is a very expressive format. Where other serialization formats may only encode the data in model weights and some information about operations, pickle models mix executable code and data.

Models that use the pickle format are encoded into a series of opcodes. When the model is loaded, these opcodes are executed by a virtual machine implemented in Python's pickle module - called the "Pickle Machine". When these opcodes execute, they instantiate the model as a python object and return it to the python program.

Most of the opcodes are very simple, but they expose functionality to import and invoke Python functions and class constructors (named "callables"). These are needed to construct complex data types like models, and load the weights values from disk.

These importing and invoking opcodes also give attackers an easy primitive to execute malicious callables – for example, by importing and calling functions like os.system.

## Existing ML Pickle Security Solutions


Source: https://docs.python.org/3/library/pickle.html



The dangers of pickle are known. Official Python documentation warns users not to deserialization untrusted data.

To protect the ML mode loading problem, we have some existing defenses:

- Alternate formats, like Hugging Face's SafeTensors, are recommended over pickle. These formats are less expressive, and typically only encode weight values.
- Model scanners are used to analyze models …
- Restricted loaders …

## Existing ML Pickle Security Solutions


**Alternate formats** with less expressive representations are recommended.
- SafeTensors was designed for security.



The dangers of pickle are known. Official Python documentation warns users not to deserialization untrusted data.

To protect the ML mode loading problem, we have some existing defenses:

- Alternate formats, like Hugging Face's SafeTensors, are recommended over pickle. These formats are less expressive, and typically only encode weight values.
- Model scanners are used to analyze models …
- Restricted loaders …

## Existing ML Pickle Security Solutions


**Alternate formats** with less expressive representations are recommended.
- SafeTensors was designed for security.

**Model scanners** analyze models to identify unsafe callables applying a *denylist*.
- Hugging Face deploys multiple model scanners, but policies are incomplete.



The dangers of pickle are known. Official Python documentation warns users not to deserialization untrusted data.

To protect the ML mode loading problem, we have some existing defenses:

- Alternate formats, like Hugging Face's SafeTensors, are recommended over pickle. These formats are less expressive, and typically only encode weight values.
- Model scanners are used to analyze models …
- Restricted loaders …

## Existing ML Pickle Security Solutions


**Alternate formats** with less expressive representations are recommended.
- SafeTensors was designed for security.

**Model scanners** analyze models to identify unsafe callables applying a *denylist*.
- Hugging Face deploys multiple model scanners, but policies are incomplete.

**Restricted loaders** only permit safe callables with restrictive *allowlists*.
- The Weights Only Unpickler is default in PyTorch 2.6.



The dangers of pickle are known. Official Python documentation warns users not to deserialization untrusted data.

To protect the ML mode loading problem, we have some existing defenses:

- Alternate formats, like Hugging Face's SafeTensors, are recommended over pickle. These formats are less expressive, and typically only encode weight values.
- Model scanners are used to analyze models …
- Restricted loaders …

## Open Questions

**Q1:** Is this a problem?

**Q2:** How do we fix it?



This raised the question: are pickle models still used in the model ecosystem? Are the existing defenses, like alternate formats and restricted loaders, sufficient?

## Empirical Study

**Are pickle models used, despite alternative formats?**

- Hugging Face longitudinal study: ~2 year period (January 2023 – March 2025)



To answer this, we conducted a longitudinal study of the model ecosystem by analyzing models on Huggging Face over a two year period. We found that despite alternate formats, the number of pickle model downloads is actually *increasing*, with over 2.1 billion monthly downloads in the last month of the study, up from ~500 million in the first.

## Empirical Study

**Are pickle models used, despite alternative formats?**

- Hugging Face longitudinal study: ~2 year period (January 2023 – March 2025)
- Finding: ***pickle downloads are increasing (2.1 billion monthly downloads)***



To answer this, we conducted a longitudinal study of the model ecosystem by analyzing models on Huggging Face over a two year period. We found that despite alternate formats, the number of pickle model downloads is actually *increasing*, with over 2.1 billion monthly downloads in the last month of the study, up from ~500 million in the first.

## Empirical Study

**Are pickle models used, despite alternative formats?**
- Hugging Face longitudinal study: ~2 year period (January 2023 – March 2025)
- Finding: ***pickle downloads are increasing (2.1 billion monthly downloads)***

**Can the Weights Only Unpickler load existing pickle models?**
- Sampled and traced pickle models in our study



We then sampled models from our study to determine whether they could be loaded by the Weights Only Unpickler, and found that 15% of sampled repositories had models that could not be loaded - indicating a usability problem with the weights only unpickler.

## Empirical Study

**Are pickle models used, despite alternative formats?**

- Hugging Face longitudinal study: ~2 year period (January 2023 – March 2025)
- Finding: ***pickle downloads are increasing (2.1 billion monthly downloads)***

**Can the Weights Only Unpickler load existing pickle models?**

- Sampled and traced pickle models in our study
- Finding: among repositories with pickle models, ***15% contain pickle model that cannot be loaded by Weights Only Unpickler***



We then sampled models from our study to determine whether they could be loaded by the Weights Only Unpickler, and found that 15% of sampled repositories had models that could not be loaded - indicating a usability problem with the weights only unpickler. These 15% are downloaded ~80 million times / month

## Open Questions

**Q1:** Is this a problem?

**Q2:** How do we fix it?



So, we recognized that pickle model loading is still a security problem.

## Open Questions

**Q1:** Is this a problem?

**Q2:** How do we fix it?



We asked, how can we design a system to fix it?

## Goal

**Requirement:** prevent malicious models from executing unnecessary callables, but permit benign models to use callables necessary to load model.



To solve the continuing Pickle security problem and to account for the limitations of existing approaches, we recognize the need for solution that prevents malicious models from executing unnecessary callables, while still permitting benign models to access the callables they need.

We want to prioritize security, to block all or as many malicious models as possible, while increasing usability of the system to correctly load more benign models.

Rather than using one-size-fits-all approaches, we want to achieve our usability goals with context specific loading policies.

## Goal

**Requirement:** prevent malicious models from executing unnecessary callables, but permit benign models to use callables necessary to load model.

**Design goal:** We aim to ***prioritize security*** by blocking malicious models, but ***improve usability*** by loading more benign models without user intervention.



To solve the continuing Pickle security problem and to account for the limitations of existing approaches, we recognize the need for solution that prevents malicious models from executing unnecessary callables, while still permitting benign models to access the callables they need.

We want to prioritize security, to block all or as many malicious models as possible, while increasing usability of the system to correctly load more benign models.

Rather than using one-size-fits-all approaches, we want to achieve our usability goals with context specific loading policies.

## Goal

**Requirement:** prevent malicious models from executing unnecessary callables, but permit benign models to use callables necessary to load model.

**Design goal:** We aim to ***prioritize security*** by blocking malicious models, but ***improve usability*** by loading more benign models without user intervention.

**Insight:** We can make ***context-specific*** loading policies, rather than one-size-fits all.



To solve the continuing Pickle security problem and to account for the limitations of existing approaches, we recognize the need for solution that prevents malicious models from executing unnecessary callables, while still permitting benign models to access the callables they need.

We want to prioritize security, to block all or as many malicious models as possible, while increasing usability of the system to correctly load more benign models.

Rather than using one-size-fits-all approaches, we want to achieve our usability goals with context specific loading policies.

## Insight: Libraries Provide Trusted Specifications

**Model Saver**

```
from lib import Model, train

model = Model()
train(model, corpus)

model.save('model.bin')
```

**Model Loader**

```
from lib import Model, train

model = Model.load('model.bin')
model.predict(input)
```



The insight to our approach is that ML libraries are a shared interface between model saving applications and model loading applications.

Engineers use libraries to create and save models - these provide utility functions for training and interacting with models.

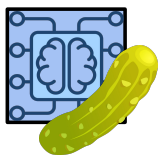
## Insight: Libraries Provide Trusted Specifications

**Model Saver**

```
from lib import Model, train

model = Model()
train(model, corpus)

model.save('model.bin')
```

**Model Loader**

```
from lib import Model, train

model = Model.load('model.bin')
model.predict(input)
```



**Imports:**
- lib.Model
- torch.Tensor
- lib.read_weights_from_file
- lib.GradDescent



The engineer will use a library API to create and save a model. If it's a pickle model, it will have some callables encoded in its representation.

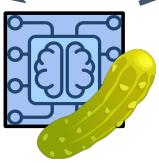
## Insight: Libraries Provide Trusted Specifications

**Model Saver**

```
from lib import Model, train

model = Model()
train(model, corpus)

model.save('model.bin')
```

**Model Loader**

```
from lib import Model, train

model = Model.load('model.bin')
model.predict(input)
```



**Imports:**
- lib.Model
- torch.Tensor
- lib.read_weights_from_file
- lib.GradDescent



To load a model, the user has to use the model loading API from the same library. Once loaded, the library provides an API for interacting with the model, like prediction tasks.

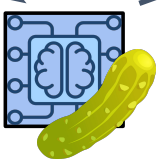
## Insight: Libraries Provide Trusted Specifications

**Model Saver**

```
from lib import Model, train

model = Model()
train(model, corpus)

model.save('model.bin')
```

**Model Loader**

```
from lib import Model, train

model = Model.load('model.bin')
model.predict(input)
```



**Imports:**
- lib.Model
- torch.Tensor
- lib.read_weights_from_file
- lib.GradDescent

Q: Should a `Model` need to import these callables?



Given a model, should it have the imports that it does, or were they placed there maliciously?

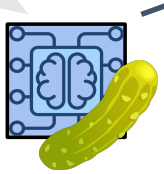
## Insight: Libraries Provide Trusted Specifications

**Model Saver**

```
from lib import Model, train

model = Model()
train(model, corpus)

model.save('model.bin')
```

**Model Loader**

```
from lib import Model, train

model = Model.load('model.bin')
model.predict(input)
```



**Imports:**
- `lib.Model`
- `torch.Tensor`
- `lib.read_weights_from_file`
- `lib.GradDescent`

Q: Should a `Model` need to import these callables?



When determining this, the user of the model does not have access to the program that created the model.

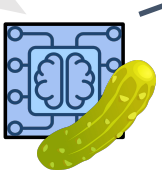
## Insight: Libraries Provide Trusted Specifications

**Model Saver**

```
from lib import Model, train

model = Model()
train(model, corpus)

model.save('model.bin')
```

**Model Loader**

```
from lib import Model, train

model = Model.load('model.bin')
model.predict(input)
```



**Imports:**
- `lib.Model`
- `torch.Tensor`
- `lib.read_weights_from`
- `lib.GradDescent`

Q: Should a `Model` need to import these callables?

A: We can look at the `Model` class definition.



But the user does know the library API that they need to load the model, and which allegedly created the model.

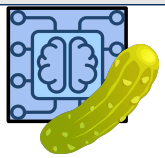
## Insight: Libraries Provide Trusted Specifications

**Model Loader**

```
from lib import Model, train

el = Model.load('model.bin')
el.predict(input)
```

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```



**Imports:**
- lib.Model
- torch.Tensor
- lib.read_weights_from
- lib.GradDescent

Q: Should a `Model` need to import these callables?

A: We can look at the `Model` class definition.



We can use the library Model class definition as a specification: we can look to it for evidence of callables that could appear in a model if the library was used to create the model.

Models that are created by other APIs or programs, including malicious ones, should not fit this specification.

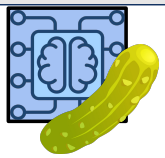
## Insight: Libraries Provide Trusted Specifications

**Model Loader**

```
from lib import Model, train

el = Model.load('model.bin')
el.predict(input)
```

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```



**Imports:**

- lib.Model
- torch.Tensor
- lib.read_weights_from
- lib.GradDescent

Q: Should a `Model` need to import these callables?

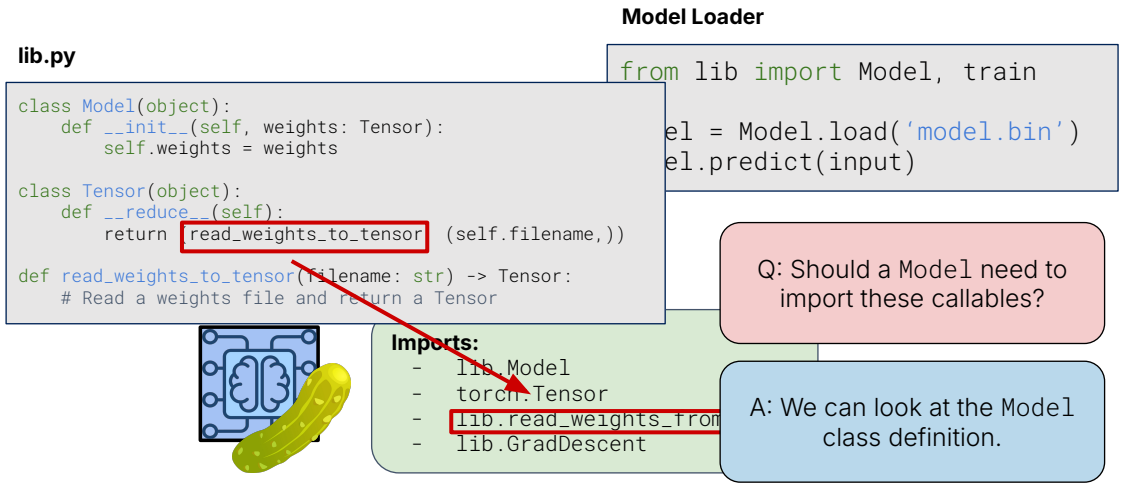
A: We can look at the `Model` class definition.



We can use the library Model class definition as a specification: we can look to it for evidence of callables that could appear in a model if the library was used to create the model.

Models that are created by other APIs or programs, including malicious ones, should not fit this specification.

## Insight: Libraries Provide Trusted Specifications

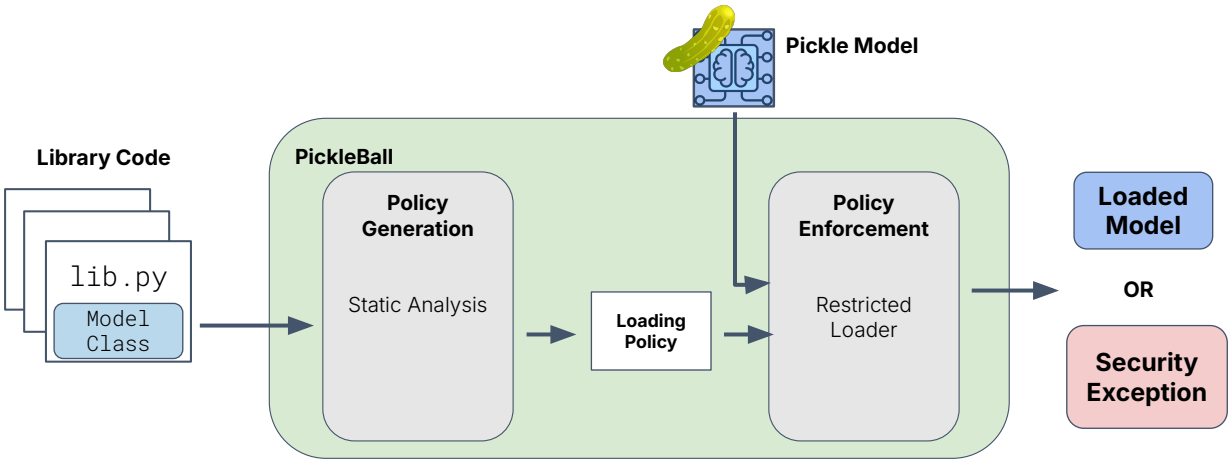





We can use the library Model class definition as a specification: we can look to it for evidence of callables that could appear in a model if the library was used to create the model.

Models that are created by other APIs or programs, including malicious ones, should not fit this specification.

## PICKLEBALL: Overview






Using this insight, we designed and implemented PickleBall, a novel approach to securely load pickle models.

PickleBall applies the standard pattern of applying static program analysis to generate a policy, which is enforced during runtime. PickleBall specializes this pattern to account for specific ML and pickle challenges.

PickleBall is implemented in two components:

1) First, policy generation: PickleBall statically analyzes the model class definition in the library source code to create a loading policy that is *specific to the library*.
2) Then, when a model needs to be loaded using the library, PickleBall enforces the policy. At the end of the model loading, PickleBall either returns the loaded model as a Python object, or raises a security exception.

## PICKLEBALL: Policy Generation

A loading policy represents all callables that could appear in any pickle program encoding a valid model.



When PickleBall generates a loading policy for a library, the policy is meant to represent all the callables that *could appear* in a pickle program that represents a valid model from the library.

## PICKLEBALL: Policy Generation

A loading policy represents all callables that could appear in any pickle program encoding a valid model.

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```

**Generated loading policy**

**Allowed Imports:**

**Allowed Invocations:**



PickleBall statically analyzes the library model class definition to generate the policy.

## PICKLEBALL: Policy Generation

A loading policy represents all callables that could appear in any pickle program encoding a valid model.

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```

**Generated loading policy**

**Allowed Imports:**
- `Model`

**Allowed Invocations:**



For a high-level summary:

PickleBall begins at the model class definition …

## PICKLEBALL: Policy Generation

A loading policy represents all callables that could appear in any pickle program encoding a valid model.

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```

**Generated loading policy**

**Allowed Imports:**
- `Model`

**Allowed Invocations:**



… and identifies what other class types are needed to instantiate a model object, based on its attribute types.

## PICKLEBALL: Policy Generation

A loading policy represents all callables that could appear in any pickle program encoding a valid model.

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```

**Generated loading policy**

**Allowed Imports:**
- `Model`

**Allowed Invocations:**



It recursively visits those types, and applies rules based on a how pickle serializes python objects, accounting for methods with special semantics for pickle, like the __reduce__ method. Its rules determine when a callable is allowed to just be imported and instantiated, or also invoked.

# PICKLEBALL: Policy Generation

A loading policy represents all callables that could appear in any pickle program encoding a valid model.

**lib.py**

```
class Model(object):
    def __init__(self, weights: Tensor):
        self.weights = weights

class Tensor(object):
    def __reduce__(self):
        return (read_weights_to_tensor, (self.filename,))

def read_weights_to_tensor(filename: str) -> Tensor:
    # Read a weights file and return a Tensor
```

**Generated loading policy**

**Allowed Imports:**
- `Model`
- `read_weights_to_tensor`

**Allowed Invocations:**
- `read_weights_to_tensor`

## Overcoming Python Analysis Challenges

**Challenge:** Python is a ***dynamic language*** but policies are generated ***statically***.



Python is a dynamic language, which makes static analysis challenging. PickleBall's policy generation is limited by some of these challenges, which it partially mitigates with its implementation.

First, missing path information can result in missing types and incomplete policies. PickleBall mitigates this with a lazy policy enforcement, that allows model loading to continue when a missing callable remains unaccessed or uninvoked. This is helpful in the ML context, where PickleBall may miss type information that represents how the model was trained after it is instantiated. If the these callables are not used during loading or inference, PickleBall skips them and continues loading.

To account for loose policies that result form over-approximate type recovery, PickleBall separates its policies into allowed imports and allowed invocations, and only adds callables to the allowed invocations when there is strong evidence that the callable needs to be invoked.

For more detailed discussion of these limitations and mitigations, I invite you to read our paper.

## Overcoming Python Analysis Challenges

**Challenge:** Python is a ***dynamic language*** but policies are generated ***statically***.

| Issue | Mitigation |
| --- | --- |



Python is a dynamic language, which makes static analysis challenging. PickleBall's policy generation is limited by some of these challenges, which it partially mitigates with its implementation.

First, missing path information can result in missing types and incomplete policies. PickleBall mitigates this with a lazy policy enforcement, that allows model loading to continue when a missing callable remains unaccessed or uninvoked. This is helpful in the ML context, where PickleBall may miss type information that represents how the model was trained after it is instantiated. If the these callables are not used during loading or inference, PickleBall skips them and continues loading.

To account for loose policies that result form over-approximate type recovery, PickleBall separates its policies into allowed imports and allowed invocations, and only adds callables to the allowed invocations when there is strong evidence that the callable needs to be invoked.

For more detailed discussion of these limitations and mitigations, I invite you to read our paper.

## Overcoming Python Analysis Challenges

**Challenge:** Python is a ***dynamic language*** but policies are generated ***statically***.

| Issue | Mitigation |
|---|---|
| Incomplete path analysis of the model creating program → **missing types** and **incomplete policies**. | PickleBall is robust to **missing** but **unused** callables through *lazy enforcement*. |



Python is a dynamic language, which makes static analysis challenging. PickleBall's policy generation is limited by some of these challenges, which it partially mitigates with its implementation.

First, missing path information can result in missing types and incomplete policies. PickleBall mitigates this with a lazy policy enforcement, that allows model loading to continue when a missing callable remains unaccessed or uninvoked. This is helpful in the ML context, where PickleBall may miss type information that represents how the model was trained after it is instantiated. If the these callables are not used during loading or inference, PickleBall skips them and continues loading.

To account for loose policies that result form over-approximate type recovery, PickleBall separates its policies into allowed imports and allowed invocations, and only adds callables to the allowed invocations when there is strong evidence that the callable needs to be invoked.

For more detailed discussion of these limitations and mitigations, I invite you to read our paper.

## Overcoming Python Analysis Challenges

**Challenge:** Python is a ***dynamic language*** but policies are generated ***statically***.

| Issue | Mitigation |
| --- | --- |
| Incomplete path analysis of the model creating program → **missing types** and **incomplete policies**. | PickleBall is robust to **missing** but **unused** callables through *lazy enforcement*. |

Handles ML usage pattern: unused training metadata



Python is a dynamic language, which makes static analysis challenging. PickleBall's policy generation is limited by some of these challenges, which it partially mitigates with its implementation.

First, missing path information can result in missing types and incomplete policies. PickleBall mitigates this with a lazy policy enforcement, that allows model loading to continue when a missing callable remains unaccessed or uninvoked. This is helpful in the ML context, where PickleBall may miss type information that represents how the model was trained after it is instantiated. If the these callables are not used during loading or inference, PickleBall skips them and continues loading.

To account for loose policies that result form over-approximate type recovery, PickleBall separates its policies into allowed imports and allowed invocations, and only adds callables to the allowed invocations when there is strong evidence that the callable needs to be invoked.

For more detailed discussion of these limitations and mitigations, I invite you to read our paper.

## Overcoming Python Analysis Challenges

**Challenge:** Python is a ***dynamic language*** but policies are generated ***statically***.

| Issue | Mitigation |
|---|---|
| Incomplete path analysis of the model creating program → **missing types** and **incomplete policies**. | PickleBall is robust to **missing** but **unused** callables through *lazy enforcement*. |
| Over-approximate type recovery → **loose policies**. | PickleBall **separates** *allowed imports* from *allowed invocations*. |

Handles ML usage pattern: unused training metadata



Python is a dynamic language, which makes static analysis challenging. PickleBall's policy generation is limited by some of these challenges, which it partially mitigates with its implementation.

First, missing path information can result in missing types and incomplete policies. PickleBall mitigates this with a lazy policy enforcement, that allows model loading to continue when a missing callable remains unaccessed or uninvoked. This is helpful in the ML context, where PickleBall may miss type information that represents how the model was trained after it is instantiated. If the these callables are not used during loading or inference, PickleBall skips them and continues loading.

To account for loose policies that result form over-approximate type recovery, PickleBall separates its policies into allowed imports and allowed invocations, and only adds callables to the allowed invocations when there is strong evidence that the callable needs to be invoked.

For more detailed discussion of these limitations and mitigations, I invite you to read our paper.

## Overcoming Python Analysis Challenges

**Challenge:** Python is a ***dynamic language*** but policies are generated ***statically***.

| Issue | Mitigation |
|---|---|
| Incomplete path analysis of the model creating program → **missing types** and **incomplete policies**. | PickleBall is robust to **missing** but **unused** callables through *lazy enforcement*. |
| Over-approximate type recovery → **loose policies**. | PickleBall **separates** *allowed imports* from *allowed invocations*. |

Handles ML usage pattern: unused training metadata

Inspired by analysis of insecure pickle implementations.

[QRS '22] Huang et al., *Pain Pickle*



Python is a dynamic language, which makes static analysis challenging. PickleBall's policy generation is limited by some of these challenges, which it partially mitigates with its implementation.

First, missing path information can result in missing types and incomplete policies. PickleBall mitigates this with a lazy policy enforcement, that allows model loading to continue when a missing callable remains unaccessed or uninvoked. This is helpful in the ML context, where PickleBall may miss type information that represents how the model was trained after it is instantiated. If the these callables are not used during loading or inference, PickleBall skips them and continues loading.

To account for loose policies that result form over-approximate type recovery, PickleBall separates its policies into allowed imports and allowed invocations, and only adds callables to the allowed invocations when there is strong evidence that the callable needs to be invoked.

For more detailed discussion of these limitations and mitigations, I invite you to read our paper.

## Evaluation: A Rigorous Dataset



To evaluate PickleBall, we constructed a dataset of libraries and models.

## Evaluation: A Rigorous Dataset

**16 ML Libraries** that load pickle models.

Represent the at-risk APIs that are compromised by malicious pickle models and **need to be protected with policies**.

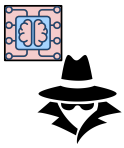


To evaluate PickleBall, we constructed a dataset of libraries and models.

## Evaluation: A Rigorous Dataset

**16 ML Libraries** that load pickle models.

Represent the at-risk APIs that are compromised by malicious pickle models and **need to be protected with policies**.



**84 malicious** models and pickle programs.

Represent all known techniques used by malicious pickle models.

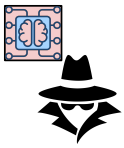


To evaluate PickleBall, we constructed a dataset of libraries and models.

## Evaluation: A Rigorous Dataset

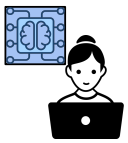
**16 ML Libraries** that load pickle models.

Represent the at-risk APIs that are compromised by malicious pickle models and **need to be protected with policies**.



**84 malicious** models and pickle programs.

Represent all known techniques used by malicious pickle models.



**252 benign** models.

Represent benign models loadable by library APIs.



To evaluate PickleBall, we needed a comprehensive dataset of libraries and models.

We built a set of 16 libraries, which represent the APIs used to create pickle models and to load pickle models. We searched huggingface for popular pickle-based models and identified the library needed to load the model. Most of the libraries extend foundational libraries like pytorch and transformers.
- [Optional - decide]: We exclude these foundational libraries because they do not produce pickle models with abnormal imports and do not have insecure loading APIs.

We created a set of 84 malicious pickle models and pickle programs. They represent all known techniques used by malicious models. It is collected from existing work that scanned all models on hugging face, and we check that its contents are representative of the models described in security reporting. We also supplemented it with two models with techniques that we found to bypass existing scanners, which we disclosed to the affected vendors.

We created a set of 252 benign pickle models. They represent the models that should be loaded without inconveniencing users. We identified models associated with each of the 16 libraries by querying Hugging Face for the libraries and their loading APIs, and filter by number of downloads.

## Evaluation: PICKLEBALL Creates Secure and Usable Policies

PICKLEBALL is compared to State of the Art solutions.

| Tool | Type | |
|---|---|---|
| ProtectAI ModelScan | Scanner | Static analysis tool integrated on Hugging Face. |
| ModelTracer | Scanner | Dynamic tracing tool proposed by Casey et al. [13]. |
| Weights Only Unpickler | Restricted Loader | Restricted loader enabled by default in PyTorch. |
| PICKLEBALL | Restricted Loader | Restricted loader with library-specific policies. |



We compare PickleBall to state of the art tools that use diverse techniques.

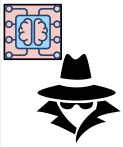
ProtectAI's ModelScan is a scanner integrated into the Hugging Face platform today to analyze models when they are uploaded to the platform. It is representative of static analysis scanners.

ModelTracer is a dynamic scanner that traces the model during loading and analyzes the trace of system calls invoked during loading. It is the only application we know of that uses dynamic tracing to analyze models.

The Weights-Only Unpickler is the default pickle loader in PyTorch, today.

## Evaluation: PICKLEBALL Creates Secure and Usable Policies

**RQ:** How well does PICKLEBALL ***prevent malicious pickle models*** from executing their payloads?



| Tool | Type | Malicious Blocked | Correct Block Rate |
|---|---|---|---|
| ProtectAI ModelScan | Scanner | 75/84 | 89.3% |
| ModelTracer | Scanner | 44/84 | 52.4% |
| Weights-Only Unpickler | Restricted Loader | 84/84 | 100% |
| PICKLEBALL | Restricted Loader | 84/84 | 100% |



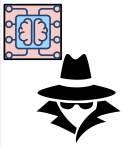
We measured how well PickleBall blocks malicious models from loading.

We find that it blocks ALL malicious models. This achieves our priority goal of security. The weights only unpickler also blocks all models, but scanners are bypassed by malicious models in our dataset.

## Evaluation: PICKLEBALL Creates Secure and Usable Policies

**RQ:** How well does PICKLEBALL ***prevent malicious pickle models*** from executing their payloads?

PICKLEBALL blocks **ALL** malicious models.



| Tool | Type | Malicious Blocked | Correct Block Rate |
|---|---|---|---|
| ProtectAI ModelScan | Scanner | 75/84 | 89.3% |
| ModelTracer | Scanner | 44/84 | 52.4% |
| Weights-Only Unpickler | Restricted Loader | 84/84 | 100% |
| PICKLEBALL | Restricted Loader | 84/84 | 100% |

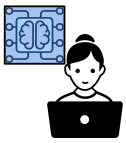


We measured how well PickleBall blocks malicious models from loading.

We find that it blocks ALL malicious models. This achieves our priority goal of security. The weights only unpickler also blocks all models, but scanners are bypassed by malicious models in our dataset.

## Evaluation: PICKLEBALL Creates Secure and Usable Policies

**RQ:** How well does PICKLEBALL correctly ***load benign pickle models***?



| Tool | Type | Malicious Blocked | Correct Block Rate | Benign Loaded | Correct Load Rate |
|---|---|---|---|---|---|
| ProtectAI ModelScan | Scanner | 75/84 | 89.3% | 236/252 | 93.6% |
| ModelTracer | Scanner | 44/84 | 52.4% | 252/252 | 100% |
| Weights-Only Unpickler | Restricted Loader | 84/84 | 100% | 157/252 | 62.3% |
| PICKLEBALL | Restricted Loader | 84/84 | 100% | 201/252 | 79.8% |



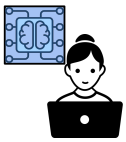
Next, we measured how well PickleBall loads benign models.

PickleBall's library specific policies and lazy enforcement allow it to load 44 more models than the Weights Only Unpickler, giving improved usability for the same secure solution. The scanners load more of the benign models without raising false alarms, but at the cost of security.

## Evaluation: PICKLEBALL Creates Secure and Usable Policies

**RQ:** How well does PICKLEBALL correctly ***load benign pickle models***?

PICKLEBALL loads **44 (17.4%)** more models than secure solutions.



| Tool | Type | Malicious Blocked | Correct Block Rate | Benign Loaded | Correct Load Rate |
|---|---|---|---|---|---|
| ProtectAI ModelScan | Scanner | 75/84 | 89.3% | 236/252 | 93.6% |
| ModelTracer | Scanner | 44/84 | 52.4% | 252/252 | 100% |
| Weights-Only Unpickler | Restricted Loader | 84/84 | 100% | 157/252 | 62.3% |
| PICKLEBALL | Restricted Loader | 84/84 | 100% | 201/252 | 79.8% |



Next, we measured how well PickleBall loads benign models.

PickleBall's library specific policies and lazy enforcement allow it to load 44 more models than the Weights Only Unpickler, giving improved usability for the same secure solution. The scanners load more of the benign models without raising false alarms, but at the cost of security.

## PICKLEBALL Provides Security and Usability

Our study shows ***pickle models are still prevalent*** in the model ecosystem, and existing security approaches are insufficient.

PICKLEBALL ***protects model users with library-specific policies*** that block all known malicious models and load more benign models than other secure alternatives.




GitHub: columbia/pickleball
Artifact: doi.org/10.5281/zenodo.16974644

**Andreas D. Kellas**
andreas.kellas@columbia.edu



In summary, today I've discussed how the pickle security threat is still prevalent in the model ecosystem, supported by our survey.

PickleBall protects model users from malicious pickle models with library-specific policies, which provide security and improved usability as a security solution. Its core idea is to use an allowlist approach that applies context to its policies, rather than a one-size-fits-all approach, and overcomes limitations of static analysis to securely improve usability with lazy enforcement.

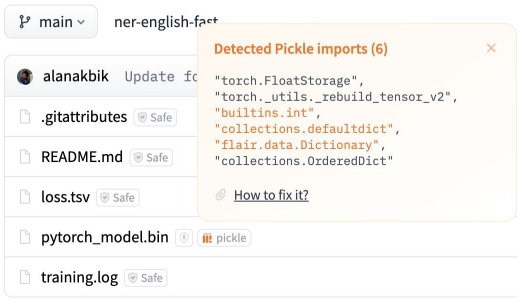
We make our artifact, including datasets, is available accessible.

Thank you for your attention, and I welcome any questions.

# Backup

# Motivating Example

**flair/ner-english-fast**



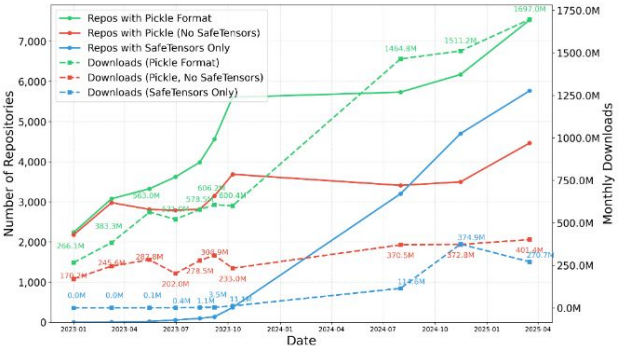

Model has 800k downloads in September 2025

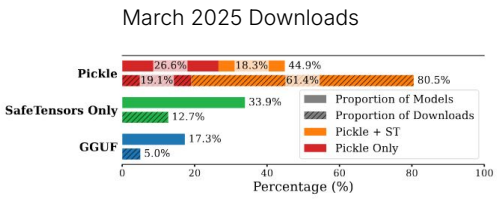
Scanners show unrecognized imports, and can be bypassed.

Weights Only Unpickler policy cannot load this model.

The user must determine themselves if the model is safe to load. ***How?***

# Pickle Model Ecosystem Study



# Benign Model Loading

| Library | Popularity | | Imports | | | Invocations | | | Loading | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Stars | Downloads | Observed | Allowed | Stub Objects | Observed | Allowed | Stub Calls | # Models | WOUp (%) | PickleBall (%) |
| CONCH [20] | 342 | 13.7K | 3 | 822 | 0 | 2 | 61 | 0 | 1 | 1 (100.0%) | 1 (100.0%) |
| FlagEmbedding [33] | 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 [107] | 1.9K | 760K | 3 | 870 | 0 | 2 | 61 | 0 | 17 | 17 (100.0%) | 17 (100.0%) |
| huggingsound [46] | 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 [23] | 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 [84] | 588 | 1.31M | 4 | 777 | 0 | 2 | 61 | 0 | 4 | 4 (100.0%) | 4 (100.0%) |
| sentence_transformers [86] | 16.4k | 204M | 5 | 1087 | 0 | 2 | 61 | 0 | 76 | 76 (100.0%) | 76 (100.0%) |
| super-image [32] | 170 | 64.9K | 3 | 1016 | 0 | 2 | 61 | 0 | 6 | 6 (100.0%) | 6 (100.0%) |
| TNER [99] | 387 | 25.0K | 4 | 769 | 0 | 2 | 61 | 0 | 4 | 4 (100.0%) | 4 (100.0%) |
| tweetnlp [12] | 341 | 80.7K | 4 | 778 | 0 | 2 | 61 | 0 | 1 | 1 (100.0%) | 1 (100.0%) |
| YOLOv5 [97] | 53.4k | 24.8K | 28 | 920 | 7 | 4 | 61 | 0 | 12 | 0 (0.0%) | 4 (33.3%) |
| YOLOv11 (ultralytics) [98] | 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% |

## SOTA Comparison

| Tool | # TP | # TN | # FP | # FN | FPR | FNR |
|---|---|---|---|---|---|---|
| ModelScan [7] | 75 | 236 | 16 | 9 | 6.3% | 10.7% |
| ModelTracer [13] | 44 | 252 | 0 | 40 | 0% | 47.6% |
| Weights-Only Unpickler [81] | 84 | 157 | 95 | 0 | 37.7% | 0% |
| PickleBall (our work) | 84 | 201 | 51 | 0 | 20.2% | 0% |

## Evaluation: PICKLEBALL Creates Secure and Usable Policies

**RQ:** Is PICKLEBALL's ***runtime overhead*** practical?

**Policy generation** (*a priori* / off-line)

PICKLEBALL **generates policies** in ***<30 seconds*** per library.

**Policy enforcement** (during loading / on-line)

PICKLEBALL **enforces policies** with a median ***1.75% runtime overhead***.



Finally, we evaluate PickleBall's runtime performance.

PickleBall generates all policies in under 30 seconds per library. This is a one-time cost per library, and once a policy is generated it can be reused to load models for that library.

During model loading, PickleBall incurs a runtime overhead of less than 2%.

## The Remaining Attack Surface

PICKLEBALL permits callables by answering "could the callable appear in a benign model"?

This is sufficient to block all discovered malicious models.

**Open question:** can permitted callables be used in *reuse-style* attacks?