Detecting Insecure Pickle Deserialization of Untrusted Data in Machine Learning Frameworks

Pinji Chen*, Yufei Ning†, Guanheng Liu‡, Jianjun Chen*✉, Minghao Zhang*,
Feiran Qin*, Qi Wang*, Haixin Duan*
*Tsinghua University, †Independent Researcher, ‡Beihang University
cpj24@mails.tsinghua.edu.cn, jianjun@tsinghua.edu.cn

***Abstract*—Pickle is Python's built-in serialization mechanism, which is widely used in machine learning (ML) frameworks for exchanging structured data. However, pickle may introduce potential security risks as deserializing untrusted data can lead to remote code execution (RCE). Prior work has primarily studied pickle security in ML supply-chain attacks, focusing on how to stealthily poison model hubs with malicious pickle files. In contrast, the vulnerabilities of insecure pickle deserialization within ML frameworks themselves during model training and inference remain largely unexplored.**

In this paper, we conduct the first systematic study of insecure pickle deserialization in ML frameworks. We introduce Picklert, a novel automated tool that can effectively detect such vulnerabilities. By applying Picklert to 50 prevailing open-source ML frameworks, we uncover 60 0-day pickle RCE vulnerabilities, including those affecting projects maintained by major organizations such as Meta, NVIDIA, and Hugging Face. Notably, 29 of these vulnerabilities cannot be detected by existing tools, and we are the first to identify a new class of vulnerabilities that arise during the model training phase. We have responsibly disclosed these issues to the affected vendors, resulting in high-severity bug bounties and the assignment of 21 CVE identifiers.

I. Introduction

With the increasing demand for resources and efficiency, modern machine learning (ML) frameworks have evolved from single-process programs into distributed systems that coordinate training and inference across multiple processes and machines [1]. This distributed design introduces numerous communication channels and requires frequent exchange of structured data such as control messages, model parameters, and intermediate results [2], [3]. To support this communication, ML frameworks always serialize structured data into a byte stream and reconstruct it by deserialization.

Python-based ML frameworks, which dominate the current ML ecosystem, typically rely on Python's built-in pickle module for serialization [4]. However, pickle may introduce security risks when applied to untrusted data sources [5]. Feeding attacker-controlled data into pickle without sanitization can directly result in remote code execution (RCE). Although this issue has been identified since 2011 [6], and several safer serializers have since been proposed [7], [8], pickle is still commonly adopted. This is because pickle can reconstruct arbitrary Python objects and invoke their methods during deserialization, whereas most other serializers are designed to reconstruct only fixed data types [9]. Since modern ML frameworks must support complex demands such as distributed task execution and result synchronization, the reliance on pickle, along with the risk of deserializing untrusted data, increases substantially. Based on our analysis of past-three-years CVEs and public bug reports [10], 24 publicly disclosed vulnerabilities were attributed to the insecure use of pickle, and 22 of them occurred in ML frameworks.

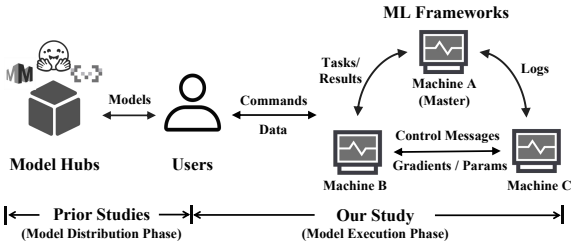


Fig. 1: Illustration of the ML lifecycle. While prior studies investigated how to poison model hubs with malicious pickle files, we focus on detecting insecure deserialization in ML frameworks during the model execution phase.

Despite the severity and pervasiveness of insecure pickle deserialization in ML frameworks, prior research [9], [11], [12], [13] has largely focused on poisoning the model supply chain, where attackers lure users into downloading malicious pickle artifacts. Conversely, as shown in Figure 1, the security risks posed by ML frameworks themselves in the model execution phase, remain largely understudied. A plausible reason is that insecure pickle deserialization has long been considered a well-understood web security issue, leading to the assumption that existing tools can effectively detect it in ML frameworks too. However, as we demonstrate in our evaluation (§V-C), existing approaches, including traditional static analysis tools (e.g., CodeQL [14]) and emerging LLM-based agents (e.g., Vulnhuntr [15] and Cursor [16]) achieve limited effectiveness in detecting such vulnerabilities. We attribute this gap to three fundamental challenges posed by the inherent characteristics

✉Corresponding author.

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2027.232783
www.ndss-symposium.org

of ML frameworks and pickle:

First, untrusted network inputs (sources) are substantially harder to identify in ML systems. ML frameworks often rely on highly customized data ingestion and communication mechanisms rather than standardized interfaces such as HTTP handlers. In many cases, these components further depend on low-level implementations [17] (e.g., sockets or native extensions via CPython's C API), introducing additional layers of abstraction and complex context. This lack of explicit and standardized network entry points makes rule-based source modeling used by static analysis tools unreliable. Similarly, LLM-based agents struggle to identify sources across such complex and layered abstractions.

Second, data propagation paths are difficult to accurately capture in ML codebases. ML frameworks, especially in distributed training settings, introduce data propagation patterns that fundamentally differ from those in conventional software. For instance, cross-device tensor operations [18] (e.g., CPU–GPU transfers) and collective communication primitives (e.g., all-reduce). Such propagation is often realized through side effects and spans multiple devices and execution contexts, rather than explicit data flow dependencies in code. This violation of the data propagation model makes it difficult for static analyzers to recover complete propagation paths. While LLMs can potentially recognize such operations, in practice, they often lack sufficient context and constraints, leading to both high false negatives and false positives.

Third, traditional sanitizer abstractions are insufficient for modeling pickle defenses. Sanitizers in static analysis are typically modeled as functions or checks that block tainted data from reaching sinks [19]. In contrast, pickle-specific defenses [20] do not intercept data during propagation. Instead, they modify the unpickling process itself by enforcing allowlists and denylists that restrict which classes and functions can be instantiated. Existing sanitizer modeling cannot fully capture the defense logic and bypassability of such defenses.

To address these challenges, we propose Picklert, a novel automated tool for detecting insecure pickle deserialization in ML frameworks. At a high level, Picklert combines the complementary strengths of static analysis and LLMs, leveraging the reliability of static methods and the strong code understanding capabilities of LLMs for complex code analysis. This design is realized through three stages. In the first stage, Picklert augments traditional rule-based source detection by utilizing semantic information from function names to identify additional candidate sources. In the second stage, Picklert uses the Vulnerable Call Chain Tree (VCCTree) extracted via static analysis as a reliable context and imposes strict state-machine-based constraints to guide the LLM's analysis process, enabling controlled vulnerable flow identification over the VCCTree. In the third stage, Picklert locates pickle defenses and extracts deserialization policies from the code logic with specific characteristics. Additionally, a gadget list (including our newly discovered gadgets leveraging CPython internal API) is applied to examine the defense effectiveness.

We evaluate Picklert on 50 popular open-source ML frameworks that have over 1,000 GitHub stars. Our evaluation shows that Picklert identifies 60 0-day RCE vulnerabilities caused by insecure pickle deserialization, achieving a precision of 82.89% and a recall of 94.03%. Compared with state-of-the-art static analysis tools (CodeQL) and LLM-based agents (Vulnhuntr and Cursor), Picklert uncovers 29 additional vulnerabilities that none of the baselines detect. Moreover, Picklert identifies a new class of pickle RCE vulnerabilities in distributed model training. Owing to the complexity of ML-specific collective primitives involved, these vulnerabilities have long remained undiscovered. This highlights Picklert's ability to perform deep semantic analysis, rather than simply relying on LLMs to match known vulnerability patterns. We have responsibly disclosed our findings to affected vendors and received high-severity bug bounties and 21 CVE identifiers.

Contributions. We make the following contributions.

- We present the first systematic study of insecure pickle deserialization in modern ML frameworks, revealing its unique vulnerability patterns and root causes.
- We propose Picklert[1], a novel automated tool that effectively detects insecure pickle deserialization in ML frameworks.
- We evaluate Picklert on 50 widely adopted ML frameworks and uncover 60 previously unknown vulnerabilities. We have responsibly disclosed these issues to affected vendors, receiving 21 CVE identifiers and high-severity bug bounties.

II. Background

A. Machine Learning Frameworks

Machine learning frameworks [1], [21] provide integrated software stacks for building, training, and deploying ML models. Such frameworks encapsulate multiple high-level APIs, enabling developers to implement complex ML workflows without managing low-level computation details.

Insecure deserialization in ML frameworks. Modern ML frameworks are inherently distributed, spanning multiple devices, processes, and machines [22], [23]. To coordinate these components, they rely on frequent exchange of structured data over various communication channels, making serialization a fundamental building block. However, the trust boundaries of these communication paths are often complex and weakly enforced. Interfaces originally designed for intra-cluster coordination may be inadvertently exposed beyond their intended scope, allowing untrusted inputs to reach internal components. Combined with developers' limited security awareness and the pervasive use of serialization mechanisms such as pickle, deserialization of untrusted data becomes common in practice, introducing severe security risks (CWE-502 [24]).

Unique characteristics of ML frameworks. First, they are not user-facing web programs with well-defined entry points, but rather complex software stacks composed of loosely coupled libraries. As a result, network inputs are often received through non-standardized interfaces or low-level APIs, making it difficult to identify attacker-controlled entry points. In addition, these interfaces are rarely specified through explicit

[1]https://github.com/Chenpinji/Picklert

interface specifications (e.g., OpenAPI), further hindering the enumeration of potential sources. Second, ML frameworks heavily rely on parallel and distributed execution semantics, including cross-device tensor operations and collective communication primitives. These behaviors introduce implicit data propagation and side effects that are difficult to capture using conventional program analysis techniques.

B. Insecure Pickle Deserialization

Pickle is Python's built-in serialization mechanism and serves as the underlying serialization engine for popular third-party libraries such as dill [25], cloudpickle [26], and joblib [27]. The primary advantage of pickle is its ability to encode and reconstruct arbitrary Python objects rather than being restricted to fixed data types, which makes it highly attractive to ML developers.

Pickle RCE vulnerability. Unfortunately, this flexibility introduces a hazardous RCE risk. A pickle byte stream encodes a sequence of opcodes that are interpreted by the unpickler, manipulating an evaluation stack and invoking operations such as object construction, function calls, and module imports [4]. Crucially, several opcodes (e.g., `GLOBAL`, `REDUCE`, and `BUILD`) enable the unpickler to resolve arbitrary callables and execute them with given arguments. Thus, deserializing a pickle payload is equivalent to executing a program in pickle's virtual machine, which could encode any malicious behavior.

Recommended pickle defenses. To mitigate pickle RCE vulnerabilities, the official Python documentation recommends preventing the loading of sensitive objects (i.e., classes or functions) during deserialization by restricting globals [20]. Specifically, the method `Unpickler.find_class` is invoked whenever a global reference is encountered in the unpickling process. By overriding this method, developers can enforce explicit allowlists or denylists to precisely control which objects are permitted to be deserialized. With the growing awareness of insecure deserialization in ML security, an increasing number of defense tools [28], [29], [30] have been developed following this principle.

III. Overview

A. Threat Model

In this paper, we consider a network attacker who can reach the vulnerable deserialization interface of an ML service, including scenarios such as (1) the service is publicly reachable, (2) the attacker resides in a shared environment (e.g., multi-tenant deployments) and can access the victim's service, or (3) the victim's internal network is reachable through vulnerabilities such as SSRF or DNS rebinding. This assumption is practical since numerous Internet-accessible deserialization interfaces are observed [31] and network exposure vulnerabilities are common in practice [32], [33]. Besides, the adversary is limited to sending network requests and cannot modify local files or poison fixed remote resources (e.g., model artifacts hosted on Hugging Face). Regarding the vulnerability scope, we focus on pickle vulnerabilities in ML frameworks that directly deserialize untrusted network data, excluding implementations that already enforce network isolation or authentication. Within this scope, our goal is to analyze and detect the most severe class of pickle vulnerabilities.

```
1  def recv_rpc_message(timeout=0):
2      msg = _CAPI_DGLRPCCreateEmptyRPCMessage()
3      status = _CAPI_DGLRPCRecvRPCMessage(timeout, msg)   #Source
4      return msg if status == 0 else None
...
5  def recv_request(timeout=0):
6      msg = recv_rpc_message(timeout)
7      req = deserialize_from_payload(req_cls, msg.data, msg.tensors)
...
8  def deserialize_from_payload(cls, data, tensors):
9      pos, nonarray_state = pickle.loads(data) #Sink
```

Fig. 2: The vulnerability we found in dmlc/dgl (14.2k stars). The source of this vulnerability is a custom C API, which cannot be predicted with predefined rules.

B. Challenges and Solutions

Prior efforts [34], [35] on detecting insecure deserialization commonly rely on static taint analysis to track untrusted data toward dangerous deserialization sinks. Trending tools like CodeQL also provide built-in support [14] for detecting insecure pickle deserialization under the same paradigm. Recently, LLM-based agents [15] have also demonstrated the ability to identify such vulnerabilities. However, applying existing methods to ML frameworks leads to three fundamental challenges:

Challenge 1: How to detect non-standardized network entry points (sources)? Unlike web applications built on frameworks such as Flask [36] or FastAPI [37] that expose standardized entry points, ML frameworks frequently implement customized communication layers through low-level network APIs. Figure 2 illustrates a representative case we identified in dmlc/dgl, where incoming RPC messages are received through a custom C API. The use of such low-level APIs is highly flexible and lacks fixed patterns, rendering rule-based source identification unreliable in ML frameworks.

Solution 1: Semantics-based source augmentation. Our key insight is that, although low-level network APIs in ML frameworks are highly flexible, they are typically wrapped by high-level functions that expose their intended semantics [17] (e.g., `recv_rpc_message`). Based on this observation, Picklert extracts function names from the code and leverages an LLM to understand which function calls are likely to correspond to network entry points. To ensure robustness, we retain conventional rule-based source detection and use this semantics-based method only for augmentation, ensuring that source coverage is never reduced.

Challenge 2: How to identify vulnerable flows under complex data propagation? In ML frameworks, distributed execution semantics introduce complicated data propagation that is not explicitly reflected in program-level data dependencies. For example, as illustrated in Figure 3, the vulnerability we found in facebookresearch/fairseq involves collective communication primitives, where tainted data is not

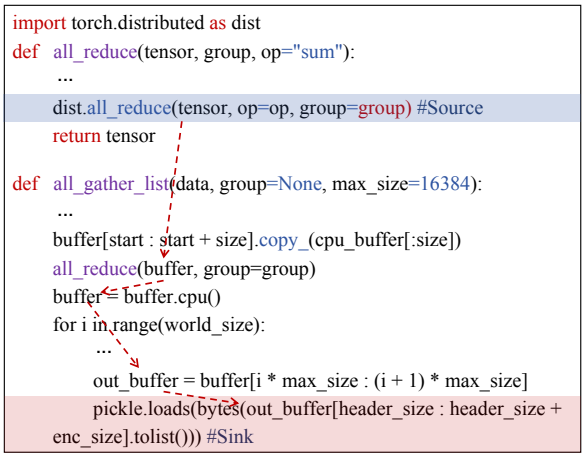


Fig. 3: The vulnerability we found in facebookresearch/fairseq (32k stars). The operation of gathering data on CPU obscures the data flow, which leads to propagation breakage.

passed through assignments. Instead, it is propagated across devices during parallel execution, temporarily materialized on CPU buffers, and later retrieved for deserialization. Such propagation patterns deviate from the data flow assumptions underlying traditional static analysis, revealing a fundamental mismatch that cannot be addressed by extending existing propagation rules.

Solution 2: VCCTree-guided stateful agent workflow. Therefore, we move to a different paradigm that leverages LLM's ability in understanding code semantics [38], [39] to identify vulnerable flows. However, directly applying LLMs to real-world ML codebases is ineffective. First, their limited context window hinders holistic analysis, making it essential to provide compact and reliable vulnerability contexts. Second, the pervasive use of pickle in ML frameworks leads to a large number of candidate chains, and relying on unconstrained LLM self-planning for long-term analysis (e.g., Vulnhuntr [15]) often results in incomplete analysis, as the model may skip steps or terminate prematurely. To address these limitations, we first introduce an efficient intermediate representation, Vulnerable Call Chain Tree (VCCTree), which is constructed via static analysis to compactly organize candidate vulnerability chains. Secondly, we design a stateful LLM-driven agent workflow, where the analysis process is strictly governed by state transitions. This design enforces step-by-step reasoning over each candidate chain, eliminating omissions caused by unconstrained planning.

Challenge 3: How to model pickle-specific defenses and assess their effectiveness? Existing static analysis tools typically model sanitizers by detecting the presence of specific filtering functions along the propagation path. However, this assumption breaks down for pickle. Pickle-specific defenses, i.e., restricting globals, operate within the unpickling procedure itself instead of during propagation. Developers override `Unpickler.find_class` to enforce module-level allowlists or denylists, thereby preventing reconstruction of sensitive objects. Although existing work [40] has explored how to bypass a given defense policy, we are not aware of existing tools that can automatically extract and model the underlying defense logic from an implementation. Furthermore, the presence of a defense does not necessarily guarantee its effectiveness, as known gadgets may have been patched or new bypass opportunities may emerge.

Solution 3: Raise-return-aligned defense modeling and CPython internal API bypass. The key to modeling pickle defenses lies in accurately capturing the deserialization policies enforced in `Unpickler.find_class`. We observe that such defenses consistently follow a common pattern: disallowed objects are rejected via exceptions, while permitted objects are returned normally. Based on this observation, Picklert statically extracts deserialization policies by identifying the raise/return logic and associated conditional checks (e.g., `in` and `not in`). To assess the robustness of these defenses, Picklert further evaluates whether known malicious gadgets satisfy the extracted constraints. As many existing gadgets have been patched, we introduce a new class of gadgets leveraging CPython internal APIs.

C. Key Findings

By applying Picklert to 50 real-world ML frameworks with over 1,000 GitHub stars, our study provides the first systematic characterization of insecure pickle deserialization in this domain, yielding three key findings:

Finding-1: The prevalence and severity of insecure pickle deserialization are critically underestimated in ML frameworks. We uncovered 60 0-day pickle vulnerabilities across 50 ML frameworks, including those maintained by major organizations such as Meta, NVIDIA, and Hugging Face. Up to now, 56 of them have been confirmed, resulting in 26 patches and 21 assigned CVEs. Notably, 17 CVEs received a CVSS score of 9.8, classified as *Critical* severity. These results reveal that some of the widely deployed real-world ML frameworks have been silently exposed to critical RCE risks.

Finding-2: Existing tools miss the majority of pickle vulnerabilities. Our experiments show that even the best-performing baseline misses 41 vulnerabilities, and 29 confirmed vulnerabilities are missed by all tools. Our ablation study further reveals the root causes: prior rule-based source identification captures only 12 of 55 valid sources, failing to cover the non-standardized entry points prevalent in ML frameworks. Traditional data flow analysis breaks down under distributed execution semantics, causing 39 vulnerabilities to be missed even when sources are identified. Moreover, even replacing the flow analysis with a raw LLM agent still misses 33 vulnerabilities due to the absence of an enforced workflow and suitable vulnerability context. These failures demonstrate that current approaches are highly unreliable in analyzing pickle vulnerabilities under complex ML codebases.

Finding-3: Discovered vulnerabilities stem from two recurring patterns, one of which exposes a novel attack surface in distributed ML training. We conclude two typical patterns that reflect how distinct architectural designs in

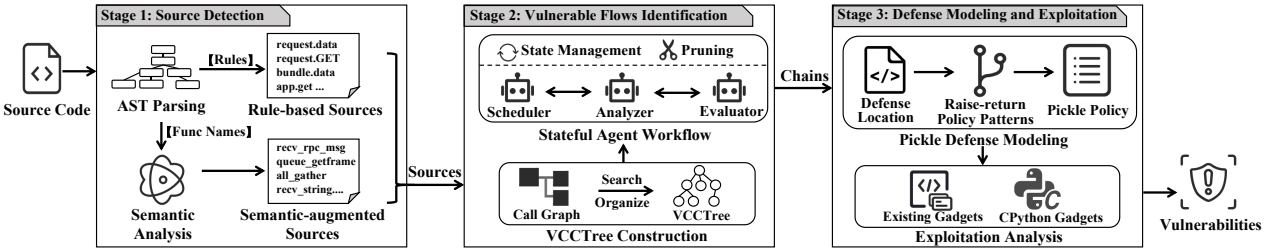


Fig. 4: Overview of Picklert

ML frameworks systematically introduce pickle vulnerabilities (details in §VI). Notably, the collaborative architecture pattern typically manifests in the distributed training phase, a setting entirely overlooked by prior work and undetected by any existing tool. These patterns can serve as practical guidance for developers and security practitioners to identify and assess pickle deserialization risks in their own ML frameworks.

IV. Picklert

In this section, we introduce Picklert, an automated tool that combines static analysis with LLM-based reasoning to detect insecure pickle deserialization in ML frameworks. As shown in Figure 4, Picklert consists of three stages:

- *Source Detection* (**§IV-A**). This stage extends conventional rule-based source identification by understanding the semantic meaning of function names to detect non-standardized network entry points in ML frameworks.
- *Vulnerable Flows Identification* (**§IV-B**). This stage first constructs a set of candidate vulnerable flows via static analysis, which are organized into a VCCTree. It then feeds these potential vulnerability contexts into a stateful agent workflow to analyze which flows correspond to real vulnerabilities.
- *Defense Modeling and Exploitation* (**§IV-C**). This stage locates pickle-specific defenses and utilizes raise/return logic to extract the deserialization policies. It then evaluates potential defense bypasses using both existing gadgets and our newly proposed CPython internal API gadgets, and reports confirmed vulnerabilities.

A. Source Detection

In the first stage, we identify potential sources by combining rule-based detection with semantic augmentation. This design preserves the reliable identification of conventional sources while enhancing coverage of non-standardized sources that are prevalent in ML frameworks. Notably, this stage is designed to maximize recall by identifying all candidate sources, rather than confirming their validity. Source validation is deferred to the Evaluator of the agent workflow (§IV-B2) in Stage 2.

Rule-based sources detection. Although ML systems are distinct from traditional web applications, some serving pipelines still reuse mainstream Python web frameworks (e.g., Flask, FastAPI, Django). Therefore, Picklert models common source patterns collected from entry-point conventions used in popular frameworks [41]. These include (1) Decorator-based endpoints (e.g., @app.post()), (2) Attribute-based access (e.g., request.data), and (3) API calls (e.g., req.get_header()). We encode these patterns as source signatures and match them statically when parsing code files.

Semantics-based sources augmentation. Nevertheless, many ML frameworks implement customized communication layers using low-level APIs (e.g., raw socket interfaces and custom C APIs) that are not well captured by predefined rules. We observe that these entry points are typically wrapped by higher-level functions whose names convey their intended semantics. Based on this insight, we adopt a two-step workflow to identify such non-standardized sources:

Step 1: Extract all function calls in the project. Picklert parses the entire codebase to enumerate all function invocations using Abstract Syntax Tree (AST). Due to Python's flexible syntax, calls may appear as simple calls (e.g., foo()) or attribute calls (e.g., module.submodule.func()). For each AST Call node: if `node.func` ∈ `ast.Name`, extract `node.func.id`. If `node.func` ∈ `ast.Attribute`, we recursively walk the attribute chain until reaching a base `ast.Name`. This ensures comprehensive coverage even for deeply nested or object-oriented API calls.

Step 2: Augment sources via function name semantics. After extracting names of all function calls, Picklert leverages an LLM to analyze which calls semantically correspond to external data ingestion. To improve the performance, Picklert applies few-shot prompting following best practices in prompt engineering [42]. We provide the model with several input–output examples, where each example contains a list of function names and the expected subset that corresponds to network entry points. As shown in Appendix B1, these examples provide the LLM with a clear task purpose. Consequently, Picklert concatenates the few-shot examples with the function calls extracted in Step 1 to form the final query. The LLM then returns the subset of functions that should be treated as potential sources for the subsequent analysis.

B. Vulnerable Flows Identification

After collecting all potential sources, we aim to identify the vulnerable flows from sources to sinks. However, the complex data propagation induced by distributed execution semantics in ML frameworks renders traditional data flow analysis ineffective. This calls for a fundamentally different analysis paradigm beyond extending existing data flow models.

High-level idea. Given the strong capabilities of LLMs in reasoning about code semantics [39], we are naturally motivated to leverage them for vulnerable flow analysis. However, directly applying LLMs is insufficient in practice due to their inherent limitations. First, their constrained context window makes it difficult to perform holistic analysis over large ML codebases, necessitating compact and informative vulnerability contexts. Second, their performance on long-horizon tasks is often unstable, leading to issues such as intent drift and step skipping. To address these limitations, we introduce a VCCTree-guided stateful agent workflow. Specifically, we construct the VCCTree via static analysis to serve as a compact representation of candidate vulnerability contexts. Building on this representation, we design a state-machine-based execution harness that explicitly regulates the LLM to iteratively analyze numerous suspicious vulnerable flows.

1) VCCTree Construction: The goal of this step is to provide LLMs with vulnerability contexts. We observe that traditional static analysis struggles to precisely identify vulnerable flows: strict propagation rules lead to false negatives, whereas relaxed rules introduce excessive false positives. However, these over-approximated flows can serve as suitable vulnerability contexts for LLM-based reasoning. Based on this insight, Picklert constructs VCCTree via a relaxed static flow analysis. The design of VCCTree is guided by two key ideas: (1) It relies on the call graph (CG), which merely captures caller–callee relationships without requiring precise data flow modeling. As a result, any source–sink pair that is functionally connected is included in the candidate set. We find this to be sufficient in practice, as our empirical evaluation shows that it does not lead to missed vulnerabilities. (2) It organizes call chains into a tree structure based on shared prefixes, enabling efficient pruning during subsequent analysis. The VCCTree construction proceeds in two steps:

Step 1: Discover all reachable call chains via bidirectional search. Following the design principles of pycg [43], Picklert constructs an interprocedural CG. It then maps sources and sinks (list in Appendix C) to their enclosing functions f_s and f_t, since the subsequent CG analysis operates at the function level. For each f_s and f_t, it checks whether a caller–callee chain exists between them on the CG using a bidirectional search algorithm. As shown in Algorithm 1, Picklert first acquires the caller closures of both f_s and f_t. Second, it identifies the set of intermediate functions that are reachable from both sides. Third, for each intermediate function, the algorithm concatenates an upward caller path from f_s with a forward callee path leading to f_t, thereby constructing a complete source-to-sink chain. This design guarantees that a valid path can be discovered regardless of whether the source function invokes the sink function, the sink function invokes the source function, or both are invoked by a common intermediate function.

Step 2: Organize paths into the VCCTree. We observe that the candidate vulnerable call chains in Step 1 may share prefixes, for instance, when the same source reaches multiple sinks. Therefore, we propose VCCTree to organize them into

Algorithm 1 Bidirectional search for source-to-sink chain

Require: Source function f_s, sink function f_t, call graph CG
Ensure: A set of paths from f_s to f_t, or $\varnothing$ if none exists
1: $A \leftarrow \textsc{CallerClosure}(f_s)$
2: $B \leftarrow \textsc{CallerClosure}(f_t)$
3: $I \leftarrow A \cap B$
4: $paths \leftarrow \varnothing$
5: **for all** $m \in I$ **do**
6: $\quad P_{\uparrow} \leftarrow \textsc{PathCaller}(f_s, m)$
7: $\quad P_{\downarrow} \leftarrow \textsc{PathCallee}(m, f_t)$
8: $\quad paths \leftarrow paths \cup \{ P_{\uparrow} \parallel P_{\downarrow}[1:] \}$
9: **end for**
10: **return** $paths$

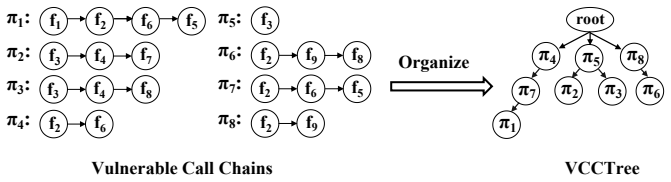


Fig. 5: Illustration of VCCTree construction

an efficient representation. Formally, given a set of call chains $\mathcal{P} = \{\pi_1, \pi_2, \ldots\}$, where each chain $\pi = \langle f_1, f_2, \ldots, f_k \rangle$, the VCCTree is a rooted directed tree $T = (V, E)$ such that (i) the root corresponds to the empty chain, (ii) each node $v \in V$ is associated with a contiguous subsequence $\mathrm{seq}(v)$ of a chain in $\mathcal{P}$, and (iii) for every chain $\pi \in \mathcal{P}$, there exists a unique leaf v with $\mathrm{seq}(v) = \pi$. For any non-root node v, its parent u satisfies that $\mathrm{seq}(u)$ is the maximal strict contiguous subsequence of $\mathrm{seq}(v)$. The VCCTree thus captures the containment relations among call chains and enables efficient pruning during analysis: if a parent node u is determined to be non-propagating, all descendant paths can be pruned. Figure 5 illustrates how a flat list of paths is deterministically transformed into a VCCTree.

2) Stateful Agent Workflow Analysis: Given the VCCTree as vulnerability contexts, Picklert employs an LLM-based workflow to analyze each candidate vulnerable flow. In contrast to existing tools that rely on the LLM's autonomous planning, Picklert schedules the analysis incrementally through a state machine. This design enforces controlled state transitions, preventing LLM's inherent issues such as step skipping and ensuring complete analysis of each candidate flow.

Formal state machine. As shown in Figure 6, we formalize the workflow as a deterministic state machine operating over three control states $S = (\textit{Scheduler}, \textit{Analyzer}, \textit{Evaluator})$. The *Scheduler* is responsible for traversing and pruning the VCCTree, and selecting the next candidate flow to analyze. The *Analyzer* retrieves relevant code snippets and determines whether the chain represents a feasible data flow from the source to the sink. The *Evaluator* further validates vulnerable flows, leveraging the full vulnerability context to assess whether the candidate source identified in Stage 1 constitutes a valid attack vector under our threat model.

To preserve analysis states across steps, we maintain a

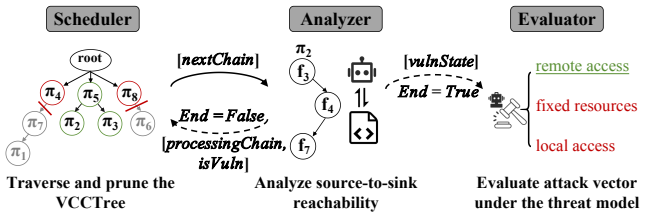


Fig. 6: Overview of our stateful agent workflow

global analysis state $\Sigma = (processingChain, nextChain, isVuln, vulnState, End)$. Among these, *processingChain* denotes the call chain currently under analysis, *nextChain* denotes the chain selected by the Scheduler for subsequent processing, *isVuln* indicates whether a vulnerable path has been confirmed, *vulnState* stores the vulnerability evidence extracted by the Analyzer and validated by the Evaluator, and *End* is the termination flag. Below, we describe the state transitions.

Scheduler state. The Scheduler receives the previous *processingChain* and *isVuln* flag and outputs the next chain to analyze. Its transition rule is formalized as:

$$\frac{C = select_next(processingChain, isVuln)}{\Sigma \xrightarrow{\text{Scheduler}} \Sigma[nextChain \leftarrow C]} \quad (1)$$

Following Eq.1, *select_next* implements our scheduling strategy over the VCCTree. The overall search follows a depth-first traversal of the tree. When a call chain (a node of the tree) is determined to be non-vulnerable (i.e., $isVuln = False$), the Scheduler immediately prunes the entire subtree rooted at that node. This pruning is sound because every leaf chain in the VCCTree strictly extends its parent chain. If a shorter subchain cannot propagate attacker-controlled data to the sink, any longer chain that contains it cannot be feasible either. Therefore, no further analysis is needed for that subtree, which significantly reduces the search space. If all call chains in the VCCTree have been ruled out, *select_next* returns *None*, indicating that the analysis should terminate.

Analyzer state. The Analyzer takes the *nextChain* selected by the Scheduler together with the current global *vulnState* as input. It performs LLM-based reasoning over the code associated with the selected call chain and updates the vulnerability verdict accordingly. Its transition rule is formalized as:

$$\frac{nextChain = C \ \wedge \ (r, s') = validate(C)}{\Sigma \xrightarrow{\text{Analyzer}} \Sigma[processingChain \leftarrow C, isVuln \leftarrow r, \ vulnState \leftarrow s', End \leftarrow (C = None)]} \quad (2)$$

In Eq. 2, *validate* implements agent-based verification of source-to-sink reachability for a given call chain. Following the prompt engineering guidelines proposed by OpenAI [44], it adopts an XML-based prompt to improve the analysis performance. The prompt is organized into four stages: (1) presenting the target call chain, (2) retrieving the relevant code snippets, (3) assessing the reachability of source-to-sink flow, and (4) formatting the response into a structured output. The output of *validate* updates the global analysis state and propagates to both the Scheduler and the Evaluator, guiding subsequent scheduling and attack vector assessment.

Evaluator state. The Evaluator is invoked after the Analyzer has finalized the vulnerability discovery. It takes the *vulnState* as input and refines this state by assessing the attack vector under the threat model. Its transition rule is formalized as:

$$\frac{End = True \ \wedge \ vulnState' = evaluate(vulnState)}{\Sigma \xrightarrow{\text{Evaluator}} \Sigma[vulnState \leftarrow vulnState', \ End \leftarrow True]} \quad (3)$$

The *evaluate* function performs source validity analysis on given vulnerable flows. Its primary task is to determine whether the corresponding sources fall under the threat model of a remote, unauthenticated attacker. Similar to the Analyzer, it adopts an XML-based prompt to guide the LLM agent. First, it acquires contexts by reading the *vulnState* and inspecting the corresponding source code. Second, it evaluates whether the source corresponds to remote open access, remote fixed resource access, or local access. Third, it examines the presence of network authentication, such as key validation. The Evaluator enriches each vulnerability in *vulnState* with attack vector attributes. The results are then written to the vulnerability report consumed by downstream defense analysis.

C. Defense Modeling and Exploitation

Although a vulnerable call chain indicates the unsafe deserialization semantics, it does not necessarily imply an exploitable vulnerability. With the growing awareness of insecure pickle deserialization in ML security, an increasing number of defenses have been proposed, many of which enforce restrictive loading policies [28], [29], [30]. This section addresses this gap by conducting defense modeling and exploitability analysis to determine whether a detected chain remains exploitable under project-specific pickle policies.

1) Raise-Return-Aligned Defense Modeling: We observe that pickle defense based on restricting globals exhibits consistent patterns, which enable the extraction of deserialization policies. As illustrated in Figure 7, such defenses are typically implemented by defining a custom class that inherits from `pickle.Unpickler` and overrides the `find_class` method (highlighted by black underlines in the figure). This provides a reliable anchor point for locating defense logic in the code. Within `find_class`, the policy is usually enforced through conditional checks (e.g., membership tests using `in` or `not in`) over module or class names, followed by explicit control-flow decisions: safe objects are returned, while disallowed ones trigger exceptions via `raise`. These *raise-return-aligned* patterns naturally encode allowlist or denylist semantics. By statically analyzing these conditionals and their associated control flow (highlighted by red underlines in the figure), Picklert extracts the sets of permitted and forbidden globals, successfully modeling pickle defense policy.

2) Defense Bypass: With the extracted defense policy, Picklert proceeds to assess defense effectiveness with bypass strategies. For allowlist-based defenses, bypass opportunities

```
safe_modules = {"numpy", "builtins", "collections"}
class RestrictedUnpickler(pickle.Unpickler):
    def find_class(self, module, name):
        package_name = module.split(".")[0]
        if package_name in safe_modules:
            return super().find_class(module, name)
        else:
            raise pickle.UnpicklingError("forbidden")
```

Fig. 7: Typical pickle defense patterns. Black underlines leave cues to identify defense locations, while red underlines highlight control-flow cues to capture policy enforcement.

are generally limited. However, overly coarse policies, e.g., allowing an entire module such as `numpy` rather than specific classes, may still enable abuse of functions within the permitted namespace. In such cases, Picklert directly uses existing payloads [13] to test whether the allowed surface suffices for malicious effects under the enforced allowlist. For denylist-based defenses, the dominant implementations (e.g., picklescan [28] and modelscan [29]) aggressively block known dangerous Python built-in functions and widely used third-party gadgets. Nevertheless, we uncover an overlooked class of gadgets, namely CPython internal API gadgets, which are not covered by any existing denylists.

Novel CPython internal API-based gadgets. In Python, identifiers with a leading underscore (e.g., _func, _module) conventionally denote non-public or internal interfaces. While such APIs are not intended for direct use by application developers, they remain fully accessible at runtime and can be invoked by attackers during deserialization. As a result, these internal interfaces form a previously underexplored attack surface for bypassing pickle defenses.

To identify exploitable gadgets leveraging CPython internal APIs, we first collect all CPython internal APIs with leading-underscore names as candidates. Second, for each candidate API, we analyze whether its execution may trigger security-critical behaviors. For APIs implemented at the Python level, we inspect whether they directly invoke system-level operations (e.g., `os.execv`) or enable indirect execution through reflective mechanisms such as `getattr`. For APIs implemented in C, we trace their exposed Python interfaces into the CPython source code to identify calls to sensitive primitives, including system calls (e.g., `fork`) and interpreter-level execution mechanisms (e.g., `PyEval_EvalCode`). As summarized in Table I, we finally identify 7 novel deserialization gadgets rooted in CPython internal APIs that lead to arbitrary command execution. We responsibly disclosed these findings to the ProtectAI Model Format Vulnerability (MFV) program [45] and the maintainers of picklescan. Following our disclosure, ProtectAI awarded us bug bounties, and picklescan released security updates accordingly, confirming the validity of the newly identified gadgets.

TABLE I: Novel gadgets leveraging CPython internal APIs

Novel Gadgets	Dangerous Execution Primitives
_posixsubprocess.fork_exec	fork/vfork
_winapi.CreateProcess	winapi
_interpreters.run_string	PyEval_EvalCode
_bootsubprocess.check_output	os.system
_bootsubprocess.Popen	os.execv/execve
_operator.methodcaller	getattr
_operator.attrgetter	getattr

V. Evaluation

A. Experimental Setup

Implementation. We implemented Picklert in over 8,000 lines of Python code. All static analyses are built on Python's native AST infrastructure. All LLMs used in Picklert are based on the DeepSeek API [46], owing to its open-source nature and strong performance in code analysis tasks. In stage 2, we use opencode [47] to build LLM-based agents, enabling them to invoke tools (e.g., reading source code files) to perform complex tasks. To manage agent workflows and intermediate states, we leverage Apache Burr [48] as the state management framework. Our experiments were conducted on a Linux server equipped with two Intel Xeon Gold 6330 CPUs and 512 GB of RAM. Regarding the cost, analyzing 50 open-source ML projects required approximately 16.8 hours, incurring about $17.3 in DeepSeek API usage.

Evaluation dataset. Our evaluation dataset consists of 50 open-source machine learning framework projects collected from GitHub, which are summarized in Appendix Table V. These projects were selected through a systematic filtering process to ensure relevance and representativeness. First, we queried GitHub using keywords (e.g., ML, AI, LLM) related to machine learning to obtain a pool of candidate repositories. Second, we only retained projects with more than 1,000 GitHub stars to ensure popularity. Third, we excluded repositories that were primarily for tutorial or test cases. Finally, we required that the source code should contain at least one common Python deserialization sink, such as pickle, dill, or their variants, ensuring relevance to deserialization analysis.

Our experiments aim to answer the following research questions:

- **RQ1:** How effective is Picklert in detecting insecure pickle deserialization within ML frameworks?
- **RQ2:** What is the performance of Picklert when compared with existing tools?
- **RQ3:** How does each component of Picklert contribute to its overall effectiveness?

B. RQ1: Picklert Effectiveness

In this section, we evaluate the effectiveness of Picklert by examining its ability to discover real-world vulnerabilities.

Overall results. We instantiated Picklert on the 50 ML frameworks in our evaluation dataset. As shown in Table II, Picklert identified 63 insecure pickle deserialization vulnerabilities across 32 ML frameworks. Each of these vulnerabilities

TABLE II: Insecure pickle deserialization vulnerabilities identified by Picklert

ML Frameworks	# Stars	# Vulns	Impact	Vulnerability Status	Positive Feedback
vllm-project/vllm	63,434	2*	RCE	Confirmed	
facebookresearch/fairseq	31,954	2	RCE	Confirmed	Add security advisory
microsoft/unilm	21,837	1	RCE	Confirmed	Arrange a fix
sgl-project/sglang	20,252	8	RCE	Confirmed	CVE-2025-6XXX5 CVE-2025-6XXX6
huggingface/lerobot	19,537	2	RCE	Confirmed	Arrange a fix
volcengine/verl	16,126	1	RCE	Pending	
NVIDIA-NeMo/NeMo	16,121	1	RCE	Confirmed	CVE-2025-3XXX8
kvcache-ai/ktransformers	15,785	1	RCE	Confirmed	CVE-2025-3XXX3
dmlc/dgl	14,141	5	RCE	Confirmed	CVE-2025-2XXX8
agentscope-ai/agentscope	13,948	1	RCE	Confirmed	High-severity bounty&CVE-2024-XXX2
pipecat-ai/pipecat	8,958	1	RCE	Confirmed	CVE-2025-6XXX3
NVIDIA/apex	8,846	1	RCE	Confirmed	CVE-2025-2XXX5
automl/auto-sklearn	8,003	1	RCE	Confirmed	CVE-2025-6XXX0
Upsonic/Upsonic	7,682	5	RCE	Confirmed	CVE-2025-5XXX5 CVE-2025-5XXX6
EleutherAI/gpt-neox	7,337	1	RCE	Pending	
InternLM/lmdeploy	7,273	2	RCE	Confirmed	CVE-2025-5XXX3
adap/flower	6,429	1	RCE	Confirmed	Bounty and CVE committed
ModelTC/LightLLM	3,731	3	RCE	Confirmed	Arrange a fix
opendilab/DI-engine	3,546	1	RCE	Confirmed	CVE-2025-6XXX4
microsoft/tensorwatch	3,465	1*	RCE	Confirmed	
PaddlePaddle/PARL	3,428	7	RCE	Confirmed	High-severity bounty
inclusionAI/AReaL	3,018	2	RCE	Confirmed	CVE-2025-6XXX0
IntelLabs/nlp-architect	2,935	1	RCE	Pending	
fugue-project/fugue	2,123	1	RCE	Confirmed	CVE-2025-6XXX3
dask/distributed	1,654	2	RCE	Confirmed	
googlecreativelab/teachablemachine-community	1,654	1	RCE	Pending	
PaddlePaddle/PaddleSlim	1,610	2	RCE	Confirmed	High-severity bounty
qinxuye/cola	1,507	1	RCE	Confirmed	CVE-2025-6XXX1
lsdefine/simple_GRPO	1,448	1	RCE	Confirmed	CVE-2025-6XXX9
Equim-chan/Mortal	1,289	2	RCE	Confirmed	CVE-2025-6XXX7 CVE-2025-6XXX8
ShannonAI/service-streamer	1,244	1	RCE	Confirmed	CVE-2025-6XXX0
marsupialtail/quokka	1,184	1	RCE	Confirmed	CVE-2025-6XXX5

[1] $*$ denotes that the vulnerabilities were previously reported. All other 60 vulnerabilities are newly discovered issues identified in this work.

enables remote attackers to achieve arbitrary code execution (RCE). Notably, 10 of the affected projects have more than 10K GitHub stars and include open-source frameworks maintained by major organizations such as Meta, NVIDIA, Microsoft, and Hugging Face, highlighting the prevalence of insecure pickle deserialization in real-world ML frameworks. More importantly, 60 out of the 63 vulnerabilities are previously unknown 0-day issues, demonstrating Picklert's strong capability in discovering new vulnerabilities at scale.

Vulnerability disclosure. Upon discovering these vulnerabilities, we responsibly disclosed all findings following the disclosure guidelines specified by each project's bug bounty program or GitHub Security Advisory process. To date, maintainers of 28 ML frameworks have acknowledged our reports and confirmed the vulnerabilities. Some vulnerabilities have already been promptly patched, while for others, we are still actively collaborating with the affected vendors to facilitate understanding and remediation of their vulnerabilities. As of now, our disclosures have resulted in 21 assigned CVE identifiers and approximately $1,500 in total bug bounty rewards. Due to the severe impact of insecure deserialization, 17 of the assigned CVEs received a CVSS score of 9.8 and were classified as Critical, while all reported CVEs have CVSS scores exceeding 7.8. These results underscore the substantial security risks that insecure deserialization poses to the modern ML framework ecosystem and also demonstrate the effectiveness of our Picklert.

False Positive (FP) analysis. We manually verified that 13 cases are false positives, which can be grouped into three categories: (1) *Sources are not remotely controllable (9/13)*. In these cases, the identified data sources are constrained to local-only communication, such as sockets bound to 127.0.0.1 or intra-process channels (e.g., inproc://), which prevents exploitation by remote attackers. We attribute these false positives to the fact that concrete binding values (e.g., IP addresses or transport modes) are sometimes defined in configuration files or separate modules, making it difficult for the agent to precisely infer remote reachability. (2) *Tainted data does not reach the pickle sink (2/13)*. These false positives are caused by the hallucination of LLM agents in long propagation chains, where attacker-controlled data reaches the sink function but is not used as the argument of the pickle deserialization call. A representative example is found in macanv/BERT-BiLSTM-CRF-NER, where a Worker receives a variable msg from a Server and subsequently invokes pickle deserialization. However, the attacker only controls msg[1], whereas the deserialized object corresponds to msg[2], which is not attacker-controlled. (3) *Unmodeled project-specific defenses (2/13)*. In two false positives from LazyAGI/LazyLLM, developers implemented customized security contexts that enforce authentication checks for network requests. Requests lacking valid credentials are rejected with a 401 response before reaching the deserialization logic. Such defenses are highly flexible and context-dependent, and are not explicitly modeled in Picklert's current defense analysis, thereby leading to false positives.

False Negative (FN) analysis. To assess false negatives,

TABLE III: Comparison of Picklert with baselines

	TP	FP	FN	Precision	Recall	F1
Picklert (Ours)	**63**	**13**	**4**	**82.89%**	**94.03%**	**88.11%**
CodeQL [14]	11	26	56	29.73%	16.42%	21.16%
Vulnhuntr [15]	10	30	57	25.00%	14.93%	18.70%
Cursor [16]	26	27	41	49.06%	38.81%	43.34%

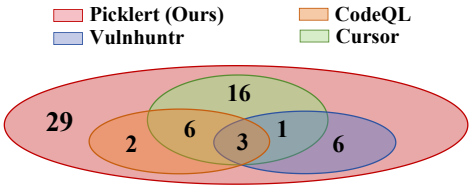


Fig. 8: Vulnerability coverage comparison between Picklert and baseline tools. Picklert identifies 29 vulnerabilities that cannot be detected by all baselines.

we first collected historical pickle vulnerabilities of the ML frameworks in our dataset from CVE records, GitHub issues, and security advisories. This results in 19 historical vulnerabilities from 15 CVEs, 1 GitHub issue, and 3 security advisories. Second, to reduce the chance of missing previously unknown vulnerabilities, we invite three security experts (each with five years of experience in vulnerability discovery) to conduct a best-effort manual audit of the dataset. They examined every potential deserialization sink (e.g., pickle.load) and manually verified whether the deserialized data could originate from an untrusted source. Consequently, we identified 4 false negatives, which fall into two categories. (1) *Missed source patterns (2/4).* These cases involve data sources from traditional web frameworks (e.g., http.server) that were accessed via request attributes and not covered by our predefined source rules. Although these two vulnerabilities can be detected after incorporating the missing source patterns, we still report them as false negatives for completeness. This observation highlights a fundamental limitation of rule-based source detection, namely its difficulty in exhaustively enumerating all possible source patterns. (2) *Highly customized gRPC communication (2/4).* These cases involve communication implemented via gRPC, where the corresponding sources are highly customized and difficult to capture. Moreover, they do not manifest as explicit function calls in the code. As a result, Picklert cannot detect them through semantics-based analysis of function names. We acknowledge this as a current limitation of our approach.

C. RQ2: Comparison with Existing Approaches

In this section, we compare Picklert with three baselines, CodeQL [14], Vulnhuntr [15], and Cursor agent [16]. CodeQL is a widely adopted static analysis framework and serves as a major baseline in prior works on detecting security vulnerabilities [49], [50], [51]. Vulnhuntr is an open-source LLM-driven vulnerability detection system, notable for discovering the first AI-discovered 0-day vulnerability, and is also capable of identifying insecure pickle deserialization. Cursor is a commercial LLM-based agent and has demonstrated strong performance in code analysis and vulnerability discovery tasks [52].

Baseline setup. For CodeQL, we directly employ its built-in Python query suite for detecting insecure deserialization vulnerabilities corresponding to CWE-502. The other two LLM-based tools are configured to use the same LLM API as Picklert to ensure fairness. All tools are evaluated with equivalent computational resources.

Comparison results. Table III summarizes the performance of Picklert compared with CodeQL, Vulnhuntr, and Cursor on detecting insecure pickle deserialization vulnerabilities. Picklert significantly outperforms all baselines, achieving a precision of 82.89% and a recall of 94.03%, whereas others exhibit substantially lower precision (29.73%, 25.00%, and 49.06%) and recall (16.42%, 14.93%, and 38.81%). In terms of coverage, as shown in Figure 8, Picklert subsumes all vulnerabilities detected by other methods. Notably, 29 confirmed vulnerabilities are missed by all baseline tools but successfully identified by Picklert. This gap highlights the limitations of existing approaches and demonstrates Picklert's state-of-the-art ability to uncover pickle deserialization vulnerabilities in complex, real-world ML frameworks.

Lessons from baseline analysis. By analyzing the missed vulnerabilities across all baselines, we derive two key lessons.

Lesson-1: Static analysis fails due to a mismatch between its underlying assumptions and the characteristics of ML frameworks. CodeQL relies on predefined source definitions and data-flow propagation models to establish source-to-sink connectivity. However, these assumptions are poorly aligned with ML frameworks, where sources are non-standardized, and data propagation deviates from conventional programming patterns due to distributed execution semantics. In our dataset, only 15 vulnerabilities originate from 12 standard web framework sources. Among these, CodeQL successfully reports 11 vulnerabilities through 8 sources, but it misses the remaining 56 primarily due to failures in source identification. Additionally, we manually provide CodeQL with all sources in the ablation study (§V-D2) to evaluate its data-flow propagation in isolation. It still misses 39 vulnerabilities. This result highlights a fundamental limitation of traditional static analysis tools when applied to ML frameworks.

Lesson-2: LLM-based approaches lack reliability without a dedicated analysis harness. Vulnhuntr adopts a lightweight paradigm that first performs static analysis to locate network-related files and then relies on the LLM to iteratively retrieve context and infer vulnerabilities. While effective for small or self-contained programs, it struggles with large-scale ML frameworks where vulnerable source-to-sink paths span multiple files and functions. This limitation is evident in our evaluation: all 10 vulnerabilities detected by Vulnhuntr are confined within a single file, with 7 of them having source-to-sink distances of fewer than 10 lines of code. This is because it largely depends on the LLM's autonomous planning, rather than operating within a constrained analysis workflow. For Cursor, as a more advanced commercial LLM-based agent, it demonstrates stronger capabilities in planning and context handling. However, our analysis of execution logs suggests the

TABLE IV: Ablation study for five variants of Picklert

Ablated Variants	TP	FP	FN	Precision	Recall	F1
Picklert-Rule	14	0	53	100.00%	20.90%	34.57%
Picklert-Sem	50	13	17	79.37%	74.63%	76.92%
Picklert-QLChain	28	13	39	68.29%	41.79%	51.85%
Picklert-RawAgent	34	26	33	56.67%	50.75%	53.55%
Picklert-NoDef	63	17	4	78.75%	94.03%	85.71%
Picklert	63	13	4	82.89%	94.03%	88.11%

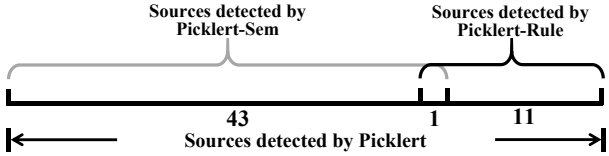


Fig. 9: Valid sources coverage comparison

absence of a strictly enforced analysis workflow. In practice, Cursor frequently skips detailed per-file inspection, relies on partial summarization, and exhibits context loss. Interestingly, we observe cases where multiple similar vulnerabilities exist, yet only a subset of them are identified due to step skipping. Moreover, it fails to detect any pickle vulnerabilities in the more complex training phase in our dataset, reflecting the difficulty of relying on LLMs to autonomously construct and reason over complicated vulnerability contexts. These findings suggest that, without suitable vulnerability context and explicit workflow planning to form a robust analysis harness, current LLMs and commercial agents are not yet reliable in uncovering complex deserialization vulnerabilities in real-world ML frameworks.

D. RQ3: Ablation Study

In this section, we conduct ablation studies to investigate the contribution of each component in Picklert. We implement five ablated variants as follows:

- **Picklert-Rule**, which removes the semantics-based source augmentation in stage 1 and retains only conventional rule-based source detection.
- **Picklert-Sem**, which removes the rule-based source detection and retains only semantics-based source detection.
- **Picklert-QLChain**, which replaces the VCCTree-guided stateful agent workflow in stage 2 with CodeQL data-flow analysis for vulnerable flows identification.
- **Picklert-RawAgent**, which replaces the VCCTree-guided stateful agent workflow in stage 2 with a raw cursor agent.
- **Picklert-NoDef**, which disables the defense modeling and exploitability analysis in stage 3.

We evaluate the five ablated variants on our evaluation dataset. The overall results of the ablation study are summarized in Table IV.

1) Assessing Source Detection: Picklert-Rule and *Picklert-Sem* are ablated variants designed to evaluate the effectiveness of source detection. Regarding the number of detected vulnerabilities, both variants miss a subset of vulnerabilities compared to the full Picklert, due to incomplete source identification. Notably, the absence of semantics-based augmentation (*Picklert-Rule*) leads to a more significant performance degradation (53 missed vulnerabilities), indicating that non-standardized sources are popular in ML frameworks. Regarding the number of sources, the distribution of detected valid sources is shown in Figure 9. Specifically, 44 sources are identified via semantic analysis of function names using the LLM, while 12 are detected through predefined rule-based matching. We also observe that the overlap between the sources identified by the two variants is minimal. This result highlights that the two detection strategies are highly complementary, and both are necessary for achieving comprehensive source coverage.

2) Assessing Vulnerable Flows Identification: Picklert identifies vulnerable flows via a VCCTree-guided stateful agent workflow. When substituting this workflow with CodeQL's taint propagation, *Picklert-QLChain* misses 39 vulnerabilities, indicating a fundamental mismatch between traditional data-flow models and the complex data propagation introduced by distributed execution semantics in ML frameworks. When replacing it with a raw Cursor agent without dedicated vulnerability contexts or stateful control, *Picklert-RawAgent* misses 33 vulnerabilities. This suggests that relying solely on the capabilities of a commercial LLM agent is insufficient. Notably, all 8 vulnerabilities in the distributed training phase remain undetected by the raw agent. We consider that this limitation is not solely due to incomplete exploration, but also reflects the difficulty of reasoning about such complex flows without structured guidance. In contrast, with VCCTree-based contexts and controlled workflow, Picklert is able to successfully identify these vulnerabilities. These results demonstrate that Picklert's design for vulnerable flow identification provides a more effective and reliable foundation for analyzing complex pickle vulnerabilities in real-world ML frameworks.

3) Assessing Defense Modeling and Exploitation: By comparing Picklert with *Picklert-NoDef*, we observe that three ML frameworks in our dataset deploy deserialization defenses by overriding `Unpickler.find_class`. Picklert's defense modeling successfully identifies all such defenses and reduces four false positives by extracting well-scoped allowlists. For exploitation analysis, Picklert further reveals that the allowlist used in uber/petastorm is overly coarse, indicating that the presence of a pickle defense does not guarantee effective protection, and Picklert can construct effective bypass gadgets. However, in this case, the sources in uber/petastorm originate from remote but fixed resources (e.g., predefined URLs). Under our threat model, such sources are not considered attacker-controllable and therefore do not constitute exploitable vulnerabilities. As restricting-globals-based pickle defenses gain increasing attention and adoption in ML frameworks [30], [53], [12], this component will become indispensable for precise vulnerability assessment.

E. Sensitivity to LLM Choice

We evaluate two alternative LLMs, GPT-5.3-Codex [54] and Kimi-2.6 [55], as drop-in replacements for DeepSeek. As shown in Appendix G, while minor differences exist

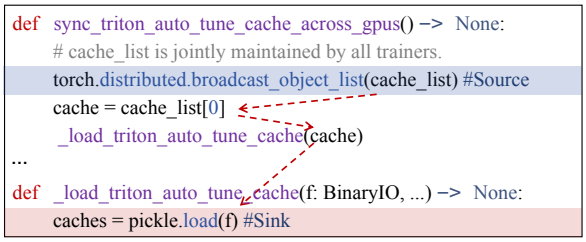


Fig. 10: Pickle RCE via shared trainer cache objects in NVIDIA/apex (8.9k stars, CVE-2025-2XXX5)

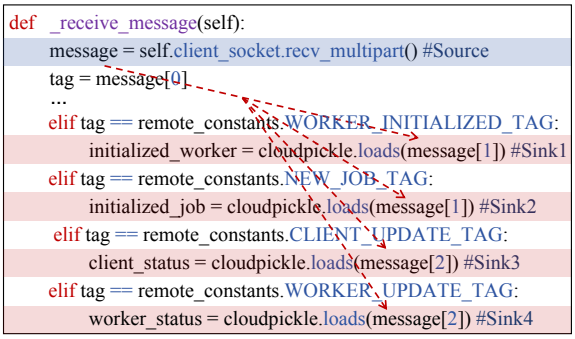


Fig. 11: Pickle RCE via master–worker messaging in PaddlePaddle/PARL (3.4k stars, high-severity bounty awarded)

across models, overall performance (e.g., F1 score) remains comparable, demonstrating that Picklert's effectiveness is not contingent on a specific LLM.

VI. Case Study

In this section, we present two novel vulnerabilities uncovered by Picklert to illustrate the representative pattern of insecure pickle deserialization in modern ML frameworks.

Case 1: Pickle RCE via shared trainer cache objects in NVIDIA/apex (8.9k stars, CVE-2025-2XXX5). In apex, all training participants collectively maintain a shared cache of Triton kernel auto-tuning results. During execution, training participants exchange cache objects via this shared cache, inserting locally generated data and retrieving objects contributed by other participants to update their local state. This synchronization process is implemented via the PyTorch collective communication primitive `torch.distributed.broadcast_object_list`.

Vulnerability details. Figure 10 illustrates the simplified vulnerable flow identified by Picklert. Specifically, `torch.distributed.broadcast_object_list` in line 3 is treated as a remote-controllable source, as the received object is supplied by another training participant rather than locally generated. The broadcasted cache object is then forwarded to `_load_triton_auto_tune_cache`, where it is deserialized via `pickle.load` without any validation. An attacker controlling one training participant can therefore inject a maliciously crafted pickle payload into the shared cache, which will be broadcast and deserialized by other participants, leading to arbitrary code execution. Notably, becoming a training participant in apex is trivial, as the training group does not enforce any authentication mechanism. A remote attacker only needs to know the master node's IP address and port to join the distributed training process. This issue was responsibly disclosed to NVIDIA, acknowledged on their official security advisory board, and assigned a CVE.

Common pattern. More broadly, this vulnerability exemplifies a recurring pattern in *ML frameworks under collaborative architecture*. In such designs, participants exchange shared caches or object lists to synchronize state, and then the received objects from others are deserialized without any sanitization. More seriously, these communication interfaces are typically implemented over raw TCP sockets and lack authentication, allowing an attacker to participate in the training process and inject malicious objects by merely knowing the master node's IP address and port. We observed the same vulnerability pattern in multiple other ML frameworks, including facebookresearch/fairseq, microsoft/unilm, and sgl-project/sglang, etc. These findings highlight a previously underexplored class of deserialization vulnerabilities unique to distributed ML training environments.

Case 2: Pickle RCE via master–worker messaging in PaddlePaddle/PARL (3.4k stars, high-severity bounty awarded). PARL adopts a master–worker architecture to coordinate training, job scheduling, and status reporting. In this design, the master node acts as a central server that receives messages from workers and clients over a network socket. Incoming messages are handled by the master's message processing logic, which dispatches different message types based on predefined tags.

Vulnerability details. Figure 11 illustrates 4 vulnerable flows in the master node's message handling routine. Specifically, the master receives external input via `self.client_socket.recv_multipart()`, which accepts arbitrary multipart messages from connected peers. Depending on the message tag, the received payload is then deserialized using `cloudpickle.loads`. An attacker can trigger arbitrary code execution on the master node by connecting to the master node and sending malicious serialized data with a specially tagged message. Notably, the PARL master node binds to `0.0.0.0` by default, exposing the service to all network interfaces. Combined with the lack of client verification, this configuration allows any remote party to establish a TCP connection and deliver malicious serialized objects, making the vulnerability remotely exploitable from the public network. This issue was responsibly disclosed to PaddlePaddle and rewarded high-severity bug bounties.

Common pattern. This case reveals another recurring vulnerability pattern in *ML frameworks under client-server architecture*. Many ML systems implement a server to exchange control signals, job descriptions, or logs, and implicitly assume that all connecting peers are trusted internal components. In practice, these services are often bound to public interfaces and lack authentication, allowing arbitrary clients to connect and supply attacker-controlled serialized data. Once an attacker can

reach the service, the direct use of pickle on network-received data enables trivial remote code execution. We observed the same pattern in other ML frameworks, including huggingface/lerobot, dmlc/dgl, and volcengine/verl, etc. These findings underscore a ubiquitous risk in ML infrastructures, where internal communication mechanisms are frequently deployed as network services without sufficient security hardening.

VII. Discussion

A. Mitigation

Many projects fundamentally rely on pickle's ability to reconstruct arbitrary Python objects for flexible data exchange, making a complete removal of pickle infeasible in some scenarios. Through responsible disclosure and discussions with multiple active vendors, including sgl-project, InternLM, and PaddlePaddle, we collaboratively distilled a practical, tiered mitigation strategy that balances security and availability.

Level 1: Replace pickle with safer serialization formats. The root cause of these vulnerabilities lies in pickle's inherently unsafe design. For ML projects that do not strictly depend on pickle's ability to deserialize arbitrary objects, we recommend replacing pickle with safer serialization formats such as JSON or MsgPack. These formats provide sufficient expressiveness for many configuration and metadata exchange tasks, while eliminating the risk of arbitrary code execution.

Level 2: Enforce class-level allowlists for pickle deserialization. Some ML frameworks require pickle's expressiveness, but only deserialize a fixed set of classes. In such cases, developers can mitigate risks by restricting globals to enforce permitted classes. This approach preserves necessary functionality while significantly reducing the attack surface by preventing the instantiation of unexpected objects.

Level 3: Constrain network exposure and make risks explicit. Certain ML frameworks deliberately support highly flexible data exchange and cannot feasibly restrict deserialized classes. For these cases, we recommend mitigating risks at the network level. If pickle is used solely for internal communication, the listening address should be strictly bound to local interfaces. When deserializing data from external sources is unavoidable, frameworks should explicitly alert the associated security risks to users. For example, by requiring an opt-in configuration flag (e.g., `I_know_pickle_rce_risk=True`). Such mechanisms ensure that insecure deserialization is a conscious, informed decision rather than an implicit default.

B. Generality and Scope

While the core techniques in Picklert can, in principle, generalize to broader Python programs and other taint-style vulnerabilities in ML frameworks, we focus on pickle within ML frameworks as a particularly high-impact setting. First, insecure pickle deserialization is one of the most serious (RCE risks) and prominent vulnerabilities in ML systems, whereas other taint-style vulnerability classes (e.g., XSS or CSRF) are less prevalent due to the absence of typical web-facing interfaces. Second, ML frameworks introduce unique challenges that fundamentally misalign with existing approaches. Although our techniques could be extended beyond this setting, the primary goal of this work is to address these ML-specific challenges and enable effective detection of pickle vulnerabilities in a high-impact yet underexplored domain.

VIII. Related Work

Pickle security. Recently, the tight coupling between ML ecosystems and pickle serialization has attracted growing attention from the community. Casey et al. [11] and Zhao et al. [9] conducted large-scale measurements of model files hosted on public model hubs, revealing that a substantial fraction of models are distributed in pickle-based formats, some of which even contain malicious payloads. In supply-chain attacks, such malicious models may pose potential security risks to downstream users. In response, platforms such as Hugging Face have integrated model scanners [56] to detect unsafe serialized model files. Subsequent studies by Zhou et al. [12] and Liu et al. [13] demonstrated that these scanners can be bypassed through a variety of techniques and identified hundreds of gadgets. While these works significantly advance the understanding of pickle security, they are primarily concerned with the model distribution phase, studying how attackers can bypass scanners and poison the model hubs with malicious pickle artifacts. Conversely, our work focuses on the model execution phase, where we investigate how to detect insecure pickle deserialization within ML frameworks. Such vulnerabilities introduce critical risks of 0-click RCE even in the absence of any poisoned model artifacts.

Detection of deserialization vulnerabilities. Numerous works have studied how to detect insecure deserialization. Most academic efforts rely on static taint analysis to identify vulnerable data flows [35], [57], [58], [59], [60] and further employ dynamic techniques, such as fuzzing, to confirm exploitability [34], [61]. However, these studies predominantly focus on Java and PHP ecosystems. Detecting insecure pickle deserialization in ML frameworks is fundamentally different due to both the flexible semantics of Python and the architectural complexity of modern ML systems. To the best of our knowledge, the work by Verma [62] is the only prior study that targets pickle deserialization vulnerabilities. However, it focuses on identifying whether online web services deserialize attacker-controlled inputs (e.g., cookies) using pickle. This scenario is substantially different from ours, as ML frameworks typically do not expose public web services. In addition, commercial analyzers, such as CodeQL, provide built-in queries for detecting insecure pickle deserialization based on static taint analysis [14]. While these approaches are effective in conventional web application settings, our experimental results show that they struggle in modern ML frameworks due to highly customized input sources and severe propagation breakage.

LLM-assisted vulnerability detection. To address the limitations of traditional static analysis, a growing number of works leverage LLMs to assist vulnerability detection. Prior studies have demonstrated the effectiveness of LLMs in diverse security analysis tasks, such as reasoning over disassembled

binaries [63], interpreting PHPDoc annotations [64], inferring error-handling specifications in C programs [65], and understanding CWE descriptions [49] to guide vulnerability detection. These works inspire us to employ LLMs as a critical building block in Picklert. Unlike prior efforts, we are the first to conduct an in-depth investigation of insecure pickle deserialization in ML frameworks. This domain-specific understanding allows us to apply LLMs in a principled and targeted manner, thereby effectively addressing the unique challenges of detecting such vulnerabilities.

IX. Conclusion

In this study, we systematically investigate insecure pickle deserialization in ML frameworks, a prevalent yet underexplored threat in ML systems. We identify key challenges that hinder existing approaches and introduce Picklert, an automated framework for detecting such vulnerabilities. By applying Picklert to 50 real-world ML frameworks, we uncover 60 previously unknown pickle RCE vulnerabilities, including cases in widely deployed systems maintained by major organizations, resulting in high-severity bug bounties and 21 CVEs. We further distill two recurring vulnerability patterns and propose a tiered mitigation strategy. We hope this work raises broader awareness of unsafe deserialization practices in ML systems and provides guidance for both researchers and practitioners to better detect and mitigate these risks.

X. Ethics Considerations

We take the utmost care of potential ethical issues. First, all vulnerability analysis was conducted in a controlled offline environment. We downloaded and analyzed open-source code locally. Our analysis did not affect any external entities or involve any user privacy. Second, all identified vulnerabilities were reported to the relevant CVE Numbering Authorities (CNAs) and affected vendors in accordance with their established disclosure procedures. In addition, we actively collaborated with project maintainers to discuss and refine a tiered mitigation strategy, which we recommended across all affected projects. We will continue to maintain communication with the affected vendors to support remediation efforts. Third, considering that some vulnerabilities are still in the process of being patched, we do not disclose their vulnerability details either in this paper or in any other public venue before the corresponding fixes are released. As a result, the publication of this paper does not introduce additional risk or harm to real-world users. Finally, the objective of our research was to understand and detect insecure pickle deserialization in ML frameworks, rather than to develop attack tools for misuse. Although releasing our artifacts may introduce potential misuse risks, we believe that the benefits outweigh these concerns. By the time this paper is published, affected vendors will have sufficient time to address the reported vulnerabilities. Moreover, making our artifacts publicly available enables developers to proactively identify insecure deserialization issues and allows future researchers to reproduce and extend our findings. Overall, our work aims to advance the understanding of insecure pickle deserialization risks in ML frameworks and to facilitate their mitigation, ultimately benefiting both developers and users of these systems.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their insightful feedback that helped improve the quality of the paper. We thank Liangsheng Yin et al. from SGLang, Qian Yao et al. from InternLM, and Hui Jiang et al. from Baidu for their active discussions and constructive comments. This work was in part supported by the NSFC #62272265. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the funding agencies.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: a system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USA: USENIX Association, 2016, pp. 265–283.

[2] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: an efficient communication architecture for distributed deep learning on gpu clusters," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USA: USENIX Association, 2017, pp. 181–193.

[3] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th USENIX Symposium on Operating Systems Design and Implementation*. USA: USENIX Association, 2014, pp. 583–598.

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

[5] Huggingface, "Pickle scanning," 2025. [Online]. Available: https://huggingface.co/docs/hub/en/security-pickle

[6] M. Slaviero, "Sour pickles," 2011. [Online]. Available: https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_Slides.pdf

[7] S. Furuhashi, "Messagepack specification," 2013. [Online]. Available: https://github.com/msgpack/msgpack/blob/master/spec.md

[8] Huggingface, "Safetensors ml safer for all," 2025. [Online]. Available: https://huggingface.co/docs/safetensors/index

[9] J. Zhao, S. Wang, Y. Zhao, X. Hou, K. Wang, P. Gao, Y. Zhang, C. Wei, and H. Wang, "Models are codes: Towards measuring malicious code poisoning attacks on pre-trained model hubs," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2087–2098.

[10] ProtectAI, "huntr - the world's first bug bounty platform for ai/ml," 2025. [Online]. Available: https://huntr.com/

[11] B. Casey, J. Santos, and M. Mirakhorli, "A large-scale exploit instrumentation study of ai/ml supply chain attacks in hugging face models," *arXiv preprint arXiv:2410.04490*, 2024.

[12] P. Zhou, "How to make hugging face to hug worms: Discovering and exploiting unsafe pickle.loads over pre-trained large model hubs," 2024. [Online]. Available: http://i.blackhat.com/Asia-24/Presentations/Asia-24-Zhou-HowtoMakeHuggingFace.pdf

[13] T. Liu, G. Meng, P. Zhou, Z. Deng, S. Yao, and K. Chen, "The art of hide and seek: Making pickle-based model supply chain poisoning stealthy again," *arXiv preprint arXiv:2508.19774*, 2025.

[14] GitHub Inc, "Codeql documentation: Deserialization of user-controlled data," 2025. [Online]. Available: https://codeql.github.com/codeql-query-help/python/py-unsafe-deserialization/

[15] D. McInerney and M. Salvati, "vulnhuntr: Zero shot vulnerability discovery using llms," 2024. [Online]. Available: https://github.com/protectai/vulnhuntr

[16] Cursor, "Cursor agent cli," 2025. [Online]. Available: https://cursor.com/en-US/blog/cli

[17] N. Christou, D. Jin, V. Atlidakis, B. Ray, and V. P. Kemerlis, "Ivysyn: automated vulnerability discovery in deep learning frameworks," in *Proceedings of the 32nd USENIX Conference on Security Symposium*. USA: USENIX Association, 2023, pp. 2383–2400.

[18] H. Y. Jhoo, S. Kim, W. Song, K. Park, D. Lee, and K. Yi, "A static analyzer for detecting tensor shape errors in deep neural network training code," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. New York, NY, USA: Association for Computing Machinery, 2022, p. 337–338. [Online]. Available: https://doi.org/10.1145/3510454.3528638

[19] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium*. USA: USENIX Association, 2005, pp. 271–286.

[20] Python Software Foundation, "pickle — python object serialization," 2025. [Online]. Available: https://docs.python.org/3/library/pickle.html#restricting-globals

[21] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019. [Online]. Available: https://arxiv.org/abs/1912.01703

[22] J. Yuan, X. Li, C. Cheng, J. Liu, R. Guo, S. Cai, C. Yao, F. Yang, X. Yi, C. Wu, H. Zhang, and J. Zhao, "Oneflow: Redesign the distributed deep learning framework from scratch," 2022. [Online]. Available: https://arxiv.org/abs/2110.15032

[23] V. J. Reddi, "Machine learning systems principles and practices of engineering artificially intelligent systems," 2025. [Online]. Available: https://mlsysbook.ai/

[24] MITRE Corporation, "Cwe-502: Deserialization of untrusted data," 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/502.html

[25] The Uncertainty Quantification Foundation, "dill package documentation," 2025. [Online]. Available: https://dill.readthedocs.io/en/latest/

[26] PiCloud, "cloudpickle," 2025. [Online]. Available: https://github.com/cloudpipe/cloudpickle

[27] Joblib developers, "Joblib: running python functions as pipeline jobs," 2025. [Online]. Available: https://joblib.readthedocs.io/en/stable/

[28] mmaitre314, "picklescan," 2025. [Online]. Available: https://github.com/mmaitre314/picklescan

[29] ProtectAI, "Modelscan: Protection against model serialization attacks," 2025. [Online]. Available: https://github.com/protectai/modelscan

[30] A. D. Kellas, N. Christou, W. Jiang, P. Li, L. Simon, Y. David, V. P. Kemerlis, J. C. Davis, and J. Yang, "Pickleball: Secure deserialization of pickle-based machine learning models," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 3341–3355.

[31] A. Lumelsky, "Shadowmq: How code reuse spread critical vulnerabilities across the ai ecosystem," 2025. [Online]. Available: https://www.oligo.security/blog/shadowmq-how-code-reuse-spread-critical-vulnerabilities-across-the-ai-ecosystem

[32] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang, "Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 239–257.

[33] L. H. Newman, "Millions of streaming devices are vulnerable to a retro web attack," 2018. [Online]. Available: https://www.wired.com/story/chromecast-roku-sonos-dns-rebinding-vulnerability

[34] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma *et al.*, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2726–2743.

[35] J. C. S. Santos, M. Mirakhorli, and A. Shokri, "Seneca: Taint-based call graph construction for java object deserialization," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649851

[36] Pallets, "Flask," 2010. [Online]. Available: https://flask.palletsprojects.com/en/stable/

[37] tiangolo, "Fastapi framework, high performance, easy to learn, fast to code, ready for production," 2025. [Online]. Available: https://fastapi.tiangolo.com/

[38] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[39] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3649828

[40] N.-J. Huang, C.-J. Huang, and S.-K. Huang, "Pain pickle: Bypassing python restricted unpickler for automatic exploit generation," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 1079–1090.

[41] V. Chen, "Awesome python restful api," 2025. [Online]. Available: https://github.com/vinta/awesome-python?tab=readme-ov-file#web-frameworks

[42] OpenAI, "Prompt engineering few-shot learning," 2025. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering#few-shot-learning

[43] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021, p. 1646–1657. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00146

[44] OpenAI, "Prompt engineering: message formatting with markdown and xml," 2025. [Online]. Available: https://platform.openai.com/docs/guides/prompt-engineering#message-formatting-with-markdown-and-xml

[45] ProtectAI, "Model file vulnerability bug bounty program," 2025. [Online]. Available: https://huntr.com/bounties

[46] DeepSeek, "Deepseek api docs," 2026. [Online]. Available: https://api-docs.deepseek.com/

[47] anomalyco, "opencode, the open source coding agent." 2026. [Online]. Available: https://github.com/anomalyco/opencode

[48] Apache Software Foundation, "Apache burr," 2025. [Online]. Available: https://burr.apache.org/

[49] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=9LdJDU7E91

[50] F. Liu, Y. Zhang, J. Luo, J. Dai, T. Chen, L. Yuan, Z. Yu, Y. Shi, K. Li, C. Zhou, H. Chen, and M. Yang, "Make agent defeat agent: automatic detection of taint-style vulnerabilities in llm-based agents," in *Proceedings of the 34th USENIX Conference on Security Symposium*. USA: USENIX Association, 2025.

[51] F. Liu, Y. Zhang, T. Chen, Y. Shi, G. Yang, Z. Lin, M. Yang, J. He, and Q. Li, "Detecting taint-style vulnerabilities in microservice-structured web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 972–990. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00137

[52] T. McPeak, "Securing our codebase with autonomous agents," 2026. [Online]. Available: https://cursor.com/cn/blog/security-agents

[53] I. Ravia, "Smashing model scanners: Advanced bypass techniques and a novel detection approach," 2025. [Online]. Available: http://i.blackhat.com/BH-USA-25/Presentations/US-25-Ravia-Smashing-Model-Scanners-Wednesday.pdf

[54] OpenAI, "Gpt-5.3-codex," 2026. [Online]. Available: https://developers.openai.com/api/docs/models/gpt-5.3-codex

[55] Kimi, "Kimi k2.6: Advancing open-source coding," 2026. [Online]. Available: https://www.kimi.com/blog/kimi-k2-6

[56] Huggingface, "Third-party scanner: Protect ai," 2025. [Online]. Available: https://huggingface.co/docs/hub/security-protectai

[57] X. Liu, H. Wang, M. Xu, and Y. Zhang, "Serdesniffer: Enhancing java deserialization vulnerability detection with function summaries," in *European Symposium on Research in Computer Security*. Springer, 2024, pp. 174–193.

[58] H. Liu and Y. Lu, "Bi-directional taint flow analysis: A high-precision static detection approach for java deserialization vulnerabilities," in *2025*

4th International Symposium on Computer Applications and Information Technology (ISCAIT), 2025, pp. 1851–1854.
[59] B. Chen, L. Zhang, X. Huang, Y. Cao, K. Lian, Y. Zhang, and M. Yang, "Efficient detection of java deserialization gadget chains via bottom-up gadget search and dataflow-aided payload construction," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 3961–3978.
[60] L. Buccioli, S. Cristalli, E. Vignati, L. Nava, D. Badagliacca, D. Bruschi, L. Lu, and A. Lanzi, "Jchainz: Automatic detection of deserialization vulnerabilities for the java language," in *Security and Trust Management*, G. Lenzini and W. Meng, Eds. Cham: Springer International Publishing, 2023, pp. 136–155.
[61] S. Rasheed and J. Dietrich, "A hybrid analysis to detect java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1209–1213. [Online]. Available: https://doi.org/10.1145/3324884.3418931
[62] A. Verma, "Insecure deserialization detection in python," 2023. [Online]. Available: https://doi.org/10.31979/etd.3yzt-6hxp
[63] P. Liu, C. Sun, Y. Zheng, X. Feng, C. Qin, Y. Wang, Z. Xu, Z. Li, P. Di, Y. Jiang, and L. Sun, "Llm-powered static binary taint analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 3, Feb. 2025. [Online]. Available: https://doi.org/10.1145/3711816
[64] Y. Ji, T. Dai, Z. Zhou, Y. Tang, and J. He, "Artemis: Toward accurate detection of server-side request forgeries through llm-assisted inter-procedural path-sensitive taint analysis," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA1, Apr. 2025. [Online]. Available: https://doi.org/10.1145/3720488
[65] P. J. Chapman, C. Rubio-González, and A. V. Thakur, "Interleaving static analysis and llm prompting," in *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. New York, NY, USA: Association for Computing Machinery, 2024, p. 9–17. [Online]. Available: https://doi.org/10.1145/3652588.3663317

APPENDIX

A. Open Science

The primary artifacts of this work include the implementation of Picklert and the code for the baseline approaches used in our evaluation. During the double-blind review process, we released all relevant code at https://anonymous.4open.science/r/Picklert-2783/ to support reproducibility and fair evaluation. Upon publication, we will publicly release the complete artifacts and replace the anonymous repository with a corresponding public repository to facilitate future research.

B. Prompt Details in Picklert

1) Prompt Details in Source Detection: At this stage, we employ few-shot prompting by supplying the model with representative input–output examples, as shown in Listing 1.

Listing 1: Few-shot examples employed in source detection

```
{
    "input": '["recv", "pipe_recv"]',
    "output": '{"result": ["recv"]}'
},
{
    "input": '["send_message", "receive_message",
    "process_data"]',
    "output": '{"result": ["receive_message"]}'
},
{
    "input": '["local_ipc_send",
    "local_ipc_receive", "log_message"]',
    "output": '{"result": []}'
},
{
    "input": '["internal_service_call",
    "external_api_request",
    "listen_for_network_messages"]',
    "output": '{"result":
    ["external_api_request",
    "listen_for_network_messages"]}'
},
{
    "input": '["connect_to_server",
    "send_heartbeat", "receive_heartbeat"]',
    "output": '{"result": ["receive_heartbeat"]}'
},
{
    "input": '["get_local_file", "read_config",
    "write_log"]',
    "output": '{"result": []}'
},
{
    "input": '["all_gather", "broadcasts",
    "all_reduce"]',
    "output": '{"result": ["all_gather",
    "broadcasts", "all_reduce"]}'
},
{
    "input": '["socket_bind", "socket_listen",
    "socket_accept", "socket_recv"]',
    "output": '{"result": ["socket_recv"]}'
},
{
    "input": '["udp_sendto", "udp_recvfrom"]',
    "output": '{"result": ["udp_recvfrom"]}'
},
{
    "input": '["tcp_sendall", "tcp_recv"]',
    "output": '{"result": ["tcp_recv"]}'
},
{
    "input": '["setup_socket", "send_data",
    "receive_response", "close_socket"]',
    "output": '{"result": ["receive_response"]}'
}
```

The system prompt used to analyze sources via function name semantics is shown in Listing 2.

Listing 2: System prompt for analyzing sources

```
You are analyzing a list of methods parsed from a
    Python file.

Your task is to:
- Analyze each method in the list.
- Determine if any of these methods are related
    to **message reception from open network or
    other ML training participants**.
  - Consider only those methods that explicitly
    deal with receiving messages over an open
    network, not including inter-process
    communication methods or those restricted to
    internal/local networks.
- If such methods exist, return them as a JSON
    list.
- If no such methods exist, return an empty JSON
    list `[]`.
- No explanation or reasoning process should be
    included in your response; provide just the
    resulting JSON list.

The input list of methods is as follows:
{input}
```

2) Prompt Details in Vulnerable Flows Identification: At this stage, we utilize XML-based prompts to instruct the Analyzer and the Evaluator to identify valid source-to-sink paths and assess their exploitability under the assumed threat

model. Listing 3 and Listing 4 present representative system prompts used by the Analyzer and the Evaluator, respectively.

Listing 3: Example system prompt for Analyzer

```
<Task>
  <instruction>
  Act as an experienced security expert.
    Following the steps and using the actual code
    and existing analysis intermediate results,
    determine whether an unsafe (e.g.
    pickle/dill/cloudpickle/joblib/torch/etc.)
    deserialization vulnerability genuinely
    exists, and output your result in a requested
    format.
  </instruction>
  <Step_by_Step_Strategy>
    <AcquireResource>
      <step1>
      Acquire Call Chain. The following call
    chain is a helpful intermediate result of
    static analysis; you need to analyze the
    vulnerabilities under the guidance of these
    call chains. The call chain is {'call chain':
    ['...'], 'sources': ['...'], 'sinks': ['...']}
      </step1>
      <step2>
      Acquire Code. Call chain mainly includes
    sources, sinks, call chain, and each line in
    the call chain is a function in a file. You
    need to grep/read the actual code (under
    {filepath}) based on the file path and
    function name to assist the analysis. At the
    same time, the source and sink may contain
    multiple, so all mentioned sources and sinks
    should be included in your analysis range.
      </step2>
    </AcquireResource>
    <AnalyzeCode>
      <step3>
      Based on the actual code, determine whether
    the unsafe input data accepted by the given
    source can flow through the call chain to the
    sink. If the unsafe input data accepted by
    the given source can flow through the call
    chain to the sink, then it is considered that
    there is a deserialization vulnerability,
    otherwise it is not considered. Do not care
    other code not in the call chain
      </step3>
    </AnalyzeCode>
  </Step_by_Step_Strategy>
  <Response>
    <response>At last, send me a JSON-format
    response only. No other content is needed.
    format is as shown below:
            '''json
            {
                'is_vulnerable': True/False,
                'source': '...',
                'sink': '...',
                'call_chain': '...',
                'code_snippet': '...',
                'description': '...'
            }
            '''
            </response>
  </Response>
</Task>
```

Listing 4: Example system prompt for Evaluator

```
<Task>
  <instruction>
  Act as an experienced security expert.
    Following the steps and using the actual code
    and vulnerability report to analyze the
    exploitability of the vulnerability.
  </instruction>
  <EvaluationStep>
    <step1>
    Acquire the Evidence: Read the vulnerability
    report in {report_path} and read/grep the
    related code in {code_path}
    </step1>
    <step2>
    Evaluate the Attack Vector: Traverse each
    vulnerability's source and determine, based
    on the actual code, which category it belongs
    to: 1. Remote Open Access. such as opening a
    socket for listening. 2. Remote Fixed
    Resource Access. such as retrieving resources
    from a fixed URL. 3. Local Access. such as
    reading local files, using user-provided
    parameters, or accessing environment
    variables.
    </step2>
    <step3>
    Evaluate the Defense: Check whether there is
    defense in the repository code, and determine
    which category it belongs to: 1. No specific
    defense. 2. Unpickler White List. 3. Key
    Validation SSH Authorization.
    </step3>
  </EvaluationStep>
  <OutputFormat>
    <makefile>
    You need to eventually write a result.json to
    record your analysis results under
    {result_path}. The result.json file contains
    the same data as {report_path}, with the only
    difference being that each vulnerability in
    report.json has been enriched with your
    analysis.
    </makefile>
    <output_format>
    For each vulnerability, add your analysis,
    with the following items: "Attack Vector":
    "..." (e.g. Remote Open Access, Remote Fixed
    Resource Access, Local Access), "Defense":
    "..." (e.g. No Specific Defense, Unpickler
    White List, Key Validation SSH Authorization)
    </output_format>
  </OutputFormat>
</Task>
```

C. Sink Patterns in Picklert

Since many third-party libraries internally rely on pickle engines, Picklert supports a set of common sink patterns for pickle-based deserialization, including:

- `pickle.load`, `pickle.loads`, `pickle.Unpickler`
- `cPickle.load`, `cPickle.loads`
- `_pickle.load`, `_pickle.loads`
- `cloudpickle.load`, `cloudpickle.loads`
- `dill.load`, `dill.loads`
- `joblib.load`
- `jsonpickle.loads`
- `torch.load`
- `zmq.Socket.recv_pyobj`
- `keras.models.load_model`

- Wildcard patterns such as `*pickle*.load*`

D. Details of Defense Extraction

To automatically extract and model pickle-specific defenses, Picklert analyzes customized `Unpickler` implementations by extracting the restriction logic enforced in the overridden `find_class` method. Specifically, Picklert first parses Python source code into abstract syntax trees and identifies classes derived from `pickle.Unpickler` or its aliases. For each customized `find_class` implementation, Picklert analyzes branch conditions involving the `module` and `name` parameters to recover the enforced restrictions. The analysis handles common patterns observed in real-world pickle defenses, including explicit collection-based policies, direct comparisons, and string matching operations. For collection-based policies, Picklert resolves constant containers such as lists, tuples, sets, and dictionaries, and converts membership checks (e.g., `(module, name) in ALLOWED_GLOBALS` or `module in BLOCKED_MODULES`) into explicit class- or module-level restrictions. It also identifies direct equality checks (e.g., `module == "os"` and `name == "system"`) and string-based matching rules such as `startswith` or substring checks, which are normalized into corresponding module or class patterns. To determine whether an extracted rule represents an allowlist or denylist, Picklert further analyzes the associated control-flow behavior: rules leading to the original `find_class` invocation are considered allowed, whereas rules leading to exception raising or rejection paths are considered forbidden. When both types of rules are present, Picklert classifies the defense as a mixed policy. Consequently, these processes extract structured deserialization policies, including the allowed and forbidden modules/classes, their corresponding matching patterns, and the overall restriction type.

E. Details on DeepSeek Model Version Update

Some vulnerabilities were first discovered by DeepSeek V3, but the final evaluation used V4 following the model update. At the early stage of our study, DeepSeek V3 identified 58 of the 63 vulnerabilities. We subsequently upgraded the model to DeepSeek V4 and extended Picklert (e.g., adding the previously missed sink zmq.Socket.recv_pyobj). In the final evaluation, we identified all 63 vulnerabilities. Among the five newly discovered vulnerabilities, two (in sglang and lmdeploy) resulted from the newly added sink, while the remaining three (two in PARL and one in dgl) may be attributed to the model upgrade.

F. Evaluation Dataset

Our evaluation dataset consists of 50 open-source ML frameworks with more than 1,000 GitHub stars. Detailed statistics are summarized in Table V.

G. Model Sensitivity Analysis

Table VI presents the performance of Picklert when instantiated with three different LLMs: DeepSeek-V4-Pro (medium reasoning effort), GPT-5.3-Codex (medium reasoning effort), and Kimi-2.6. All three models achieve comparable F1 scores (88.11%, 88.24%, and 83.44%, respectively), confirming that Picklert's effectiveness is not contingent on a specific LLM. We attribute the overall consistency across models to our stateful agent workflow, which constrains LLM behavior through strict state transitions, limiting the influence of model-specific tendencies on the final vulnerability verdict.

TABLE V: Evaluation dataset of 50 ML frameworks

ML Frameworks	Version	# Stars	# LoCs
binary-husky/gpt_academic	3.91	69,908	37,703
vllm-project/vllm	0.11.0	63,434	631,797
deepspeedai/DeepSpeed	0.18.0	40,737	216,784
ray-project/ray	2.50.1	39,902	1,086,368
facebookresearch/fairseq	0.12.3	31,954	179,214
microsoft/unilm	yoco.v0	21,837	1,197,554
sgl-project/sglang	0.5.6	20,252	596,628
huggingface/lerobot	0.4.2	19,537	75,604
volcengine/verl	0.6.0	16,126	117,419
NVIDIA-NeMo/NeMo	2.5.0	16,121	672,857
kvcache-ai/ktransformers	0.4.1	15,785	192,542
google/adk-python	1.18.0	15,357	176,225
horovod/horovod	0.28.1	14,628	78,338
dmlc/dgl	2.4.0	14,141	289,276
agentscope-ai/agentscope	0.1.7	13,948	39,679
pipecat-ai/pipecat	0.0.89	8,958	76,552
NVIDIA/apex	25.07	8,846	81,759
xorbitsai/inference	1.12.0	8,760	243,493
apache/beam	2.69.0	8,374	2,228,537
automl/auto-sklearn	0.15.0	8,003	102,755
Upsonic/Upsonic	0.55.5	7,682	74,110
EleutherAI/gpt-neox	2.0	7,337	131,201
InternLM/lmdeploy	0.10.1	7,273	144,535
adap/flower	1.17.0	6,429	327,307
macanv/BERT-BiLSTM-CRF-NER	ccf3f09	4,867	8,717
Eventual-Inc/Daft	0.6.10	4,792	383,568
ModelTC/LightLLM	1.1.0	3,731	80,044
LazyAGI/LazyLLM	0.6.3	3,551	61,158
opendilab/DI-engine	0.5.3	3,546	144,812
microsoft/tensorwatch	0.9.0	3,465	22,694
PaddlePaddle/PARL	2.2	3,428	68,881
inclusionAI/AReaL	0.3.3	3,018	151,982
Kedreamix/Linly-Talker	1.0	3,006	151,864
IntelLabs/nlp-architect	0.5.5.1	2,935	443,864
IAHispano/Applio	3.5.1	2,698	46,375
facebookresearch/habitat-lab	0.3.3	2,680	77,257
alibaba/ROLL	0.1.3	2,318	329,929
fugue-project/fugue	0.9.1	2,123	30,705
uber/petastorm	0.13.0rc0	1,868	12,339
dask/distributed	2025.10.0	1,654	112,144
googlecreativelab/teachablemachine-community	0.1.0	1,654	43,509
PaddlePaddle/PaddleSlim	2.6.0	1,610	226,444
microsoft/MarS	77e7845	1,590	6,426
qinxuye/cola	0.1.0	1,507	7,225
lsdefine/simple_GRPO	4ab94c6	1,448	2,247
NJU-PCALab/STAR	69b8bc5	1,388	32,053
Equim-chan/Mortal	edb448	1,289	18,129
ShannonAI/service-streamer	0.1.2	1,244	1,329
marsupialtail/quokka	0.3.1	1,184	132,927
uber-research/PPLM	e236b89	1,151	24,304

TABLE VI: Model sensitivity analysis of Picklert

	TP	FP	FN	Precision	Recall	F1
DeepSeek-V4-Pro	63	13	4	82.89%	94.03%	88.11%
GPT-5.3-Codex	60	9	7	86.96%	89.55%	88.24%
Kimi-2.6	63	21	4	75.00%	94.03%	83.44%