On the (In)Security of Loading Machine Learning Models

Gabriele Digregorio, Marco Di Gennaro, Stefano Zanero, Stefano Longari, Michele Carminati
Politecnico di Milano
Milan, Italy
{*gabriele.digregorio, marco.digennaro, stefano.zanero, stefano.longari, michele.carminati*}*@polimi.it*

***Abstract*—The rise of model sharing through frameworks and dedicated hubs makes Machine Learning significantly more accessible. Despite its benefits, loading shared models exposes users to underexplored security risks, while security awareness remains limited among both practitioners and developers. To enable a more security-conscious approach in Machine Learning model sharing, in this paper, we evaluate the security posture of frameworks and hubs, assess whether security-oriented mechanisms offer real protection, and survey how users perceive the security narratives surrounding model sharing. Our evaluation shows that most frameworks and hubs address security risks partially at best, often by shifting responsibility to the user. More concerningly, our analysis of frameworks advertising security-oriented settings and complete model sharing uncovered multiple 0-day vulnerabilities enabling arbitrary code execution. Through this analysis, we show that, despite the recent narrative, securely loading Machine Learning models is far from being a solved problem and cannot be guaranteed by the file format used for sharing. Our survey shows that the security narrative leads users to consider security-oriented settings as trustworthy, despite the weaknesses shown in this work. From this, we derive suggestions to strengthen the security of model-sharing ecosystems.**

1. Introduction

In recent years, the adoption of Machine Learning (ML) has grown rapidly, with advanced tools once limited to experts now accessible to a broad audience. Among other factors, this trend is driven by the rise of platforms for sharing pre-trained models [1], [2], [3], [4], [5]. Such models are often advertised as ready-to-use, lowering entry barriers and enabling practitioners with different levels of expertise to incorporate ML into their workflows.

This trend reflects the evolution of traditional software development, where open-source repositories enable the reuse and adaptation of algorithms, code, and libraries. Today, models are shared and reused, enabling a collaborative ecosystem that is reshaping how ML is practiced. However, unlike the well-studied risks of code sharing and software supply chains [6], [7], [8], [9], [10], [11], [12], the security implications of ML model sharing remain largely underexplored. Prior work on malicious code injection in models [13], [14], [15], [16] and on hub security gaps [16], [17], [18], [19] remains fragmented, which may contribute to limited awareness among users, framework developers, and sharing hubs. Zhu et al. [20] further suggest open challenges in this space by showing that TensorFlow APIs can be abused for file and network access at inference time. The growing relevance of the problem is also reflected in a recent DEF CON 33 talk [21], which highlighted persistent risks in model sharing, such as insecure pickle deserialization.

Our work provides a systematic analysis of the security landscape of ML model sharing. We evaluate major ML frameworks and sharing hubs, uncover previously unknown vulnerabilities, and expose a misalignment between perceived and actual security mechanisms. Overall, our findings emphasize the need to apply the same level of rigor to the study of model-sharing threats as is currently devoted to open-source software security at large.

We consider a threat model where attackers craft malicious model artifacts to compromise a victim's system, with arbitrary code execution as the goal. Our research is driven by the questions defined below:

RQ1. *What is the security posture of the model-sharing mechanisms adopted by popular ML frameworks and hubs?*

RQ2. *Are approaches claiming security, while offering full model object sharing, actually secure?*

RQ3. *Is the user's perception of the security posture consistent with reality, or does the security narrative affect their understanding of the sharing-associated risks?*

To answer RQ1, we analyze how the most popular ML frameworks and hubs address model sharing. We observe that while some of them do not address security at all, others provide security-oriented mechanisms based on shifting responsibility to other parts of the sharing workflow or on restricting the expressiveness of the model representation. Some go further, with certain frameworks providing complete model-sharing capabilities while explicitly promoting themselves as security-oriented, and some hubs emphasizing the same security focus through safeguards such as content scanning. This narrative is reinforced through documentation and naming choices (e.g., Keras's "safe mode" [22]), and is often justified by the use of data-based formats, assumed to reduce the risk of arbitrary code execution.

This class of security-oriented sharing frameworks and hubs led to RQ2, which motivates a vulnerability assessment of their features. In our investigation, we discovered

a remarkably low number of pre-existing *Common Vulnerabilities and Exposures* (referred to as *CVEs*) affecting these mechanisms. Through a manual reverse engineering analysis, we uncovered six 0-day vulnerabilities (i.e., previously undisclosed), each enabling arbitrary code execution. Among these, we identified the first officially recognized CVEs targeting Keras's "safe mode" [23], [24]. Collectively, these vulnerabilities challenge the widespread assumption that data-based formats (e.g., JSON) are inherently *secure* when used to share full model objects. Overall, our findings in response to RQ2 reveal a critical gap between the *security narrative* and the actual implementation.

This understanding led to RQ3. To address it, we conducted a survey targeting ML practitioners to examine how the security narratives promoted by frameworks and hubs influence user perception. The results worryingly indicate that security-oriented terminology and claims in documentation significantly shape users' sense of security.

We conclude our paper with takeaways and suggestions for users and developers, synthesized from the observations and insights obtained through our research questions.

Contributions. We make the following contributions:

- We systematize and analyze the security of ML model-sharing mechanisms, covering both framework-level and hub-level perspectives.
- We assess methods that claim security while offering full model object sharing, identifying several CVEs, and challenging the common assumption that data-based formats are inherently secure.
- We reveal a disconnect between the perceived security narrative (and the resulting community belief) and the reality through a survey on model sharing.
- We provide takeaways for the community to promote a more security-aware culture in model sharing.

Open Science. We provide all artifacts necessary to reproduce our empirical results. A repository, available at https://zenodo.org/records/19224108, contains the frozen version of our proof-of-concept (PoC) exploits, model artifacts, survey data, and analysis scripts, i.e., the exact version evaluated during the artifact evaluation process. For the updated version of the artifacts, please refer to the GitHub repository available at https://github.com/necst/security-model-sharing.

2. Sharing Machine Learning Models

ML model sharing can be examined from multiple perspectives, depending on how models are stored, distributed, and loaded. This work analyzes the security aspects of model sharing at two levels: the *framework level*, which concerns how frameworks handle model serialization and loading, and the *hub level*, which focuses on the distribution practices adopted by model-sharing platforms (*hubs*). We focus on the most widely adopted ML frameworks (Section 2.1) and hubs (Section 2.2), identifying their recommended sharing formats and practices from official documentation. Consequently, we do not consider third-party sharing mechanisms that are not officially endorsed by the frameworks and hubs analyzed in this study. This section provides the necessary background, while the security implications of model sharing are discussed in Section 3.

2.1. Framework-level Sharing

During our preliminary analysis, we observed that framework-level model sharing formats can differ both in the content they store and in the way they represent a model. With respect to the stored content, we observed that some formats include all components needed to restore a model via a single loading API, while others store only weights or configurations and require a separate model definition. We refer to the former as *self-contained* formats and to the latter as *non-self-contained* formats. Regarding model representation, some formats serialize model code objects directly (e.g., pickle), whereas others describe model structures declaratively (e.g., JSON). We refer to the former as *code-based* formats and to the latter as *data-based* formats.

We then systematically analyze these two dimensions (i.e., stored content and model representation) across the five most widely adopted ML frameworks, as identified in the 2022 Kaggle ML & DS Survey [35]: TensorFlow [26], Keras [25], PyTorch [29], scikit-learn [31], and XGBoost [33]. Notably, TensorFlow is the only one in this group that is not strictly Python-based, highlighting the dominance of Python in ML fields. According to the same survey, 88.3% of ML practitioners identified Python as their primary programming language, far surpassing all alternatives. This distribution is further confirmed by the 2024 JetBrains Python Developers Survey [36], which reports usage rates of 48% for TensorFlow, 30% for Keras, 60% for PyTorch, 67% for scikit-learn, and 22% for XGBoost.

We now analyze the model-sharing techniques suggested by the official documentation of each selected framework. Table 1 summarizes their main characteristics and security implications, discussed in Section 3.

2.1.1. Keras. Keras is a Deep Learning API [25] tightly integrated into TensorFlow [26], where it serves as the default high-level interface. Keras provides APIs [22] for saving and loading models, enabling users to preserve architecture, weights, and training configuration.

Self-contained Formats. The recommended persistence format is the `.keras` archive, which has been introduced in Keras 2.11. It is a ZIP file whose core element is a `config.json`, which describes the model's configuration and architecture in a hierarchical JSON structure. This data-based representation typically allows models to be restored without requiring prior class or function definitions. Keras provides a `safe_mode` option (introduced in v2.13) to restrict insecure deserialization during model loading [22]. Its documentation states:

"`safe_mode`: Boolean, whether to disallow unsafe lambda deserialization. When `safe_mode=False`, loading an object has the potential to trigger arbitrary code execution.

TABLE 1: Framework-level ML model sharing mechanisms and security considerations, summarizing the persistence formats documented by the analyzed ML frameworks and indicating whether they are data- or code-based, self-contained (i.e., sufficient to reconstruct the full model), and their associated security models.

Framework	Setting	File Format		Self-contained		Security model (security posture + commentary)	
		Data-based	Code-based	Weights	Model*	SO	Comments
Keras [22], [25]	`safe_mode=True`	JSON	-	●	●	✓	Blocks untrusted `Lambda` deserialization; custom objects must be registered
	`safe_mode=False`	JSON	pickle	●	●	×	Can restore models with unrestricted code; arbitrary code execution is possible by design
	Legacy	HDF5	pickle	●	●	×	Legacy format; can restore unrestricted code; arbitrary code execution is possible by design
	Weights-only	HDF5	-	●	○	✓	Model must be provided separately (imported, downloaded, or copy–pasted)
TensorFlow [26], [27], [28]	`SavedModel`	protobuf + raw data + assets	-	●	●	×	Encodes computational graphs and weights; arbitrary code execution is possible by design
	Checkpoint	raw data	-	●	○	✓	For training checkpoints, not for sharing; model must be known and defined
PyTorch [29], [30]	`weights_only=False`	-	pickle	●	●	×	Can restore models with unrestricted code; arbitrary code execution is possible by design
	`weights_only=True`	-	pickle	●	○	✓	Restricted unpickler; model must be provided separately (imported, downloaded, or copy–pasted)
scikit-learn [31], [32]	Pickle-based	-	pickle joblib cloudpickle	●	●	×	Can restore models with unrestricted code; arbitrary code execution is possible by design
	With Skops	JSON	-	●	●	✓	Allow-listed trusted types; flagged types require user review at load
	With ONNX	protobuf	-	●	◐	✓	Restricts the set of operations a model can use to implement its inference function
XGBoost [33], [34]	Model	JSON UBJSON	-	●	◐	✓	No executable code is saved; to resume training, the hyperparameters must be provided separately
	Model + hyperparams	-	pickle	●	●	×	Can restore models with unrestricted code; arbitrary code execution is possible by design

* The *Model* column is marked when the sharing format allows, in principle, to restore the complete model without manually redefining or reinstantiating it. Even when marked, some models may still require manual code definition (e.g., when they contain custom or non-standard objects).
Legend: ● = Supported, ◐ = Partially supported (see *Comments* column), ○ = Not supported, SO = Security-Oriented, ✓= Yes, ×= No, Green = mechanisms that present security-oriented features while being self-contained.

This argument is only applicable to the Keras v3 model format. Defaults to `True`." [22]

This option addresses a key risk: Keras supports `Lambda` layers, which can wrap arbitrary Python expressions as `Layer` objects. When serialized, these layers may embed Python bytecode representing custom logic. If `safe_mode` is disabled, this bytecode is deserialized and will execute at load time. In such cases, the `.keras` format becomes hybrid, combining data with executable code. Additionally, developers may allow custom, potentially *"unsafe"* types by using a specific decorator [22]. In earlier versions, complete models were saved in the HDF5 format (`.h5`). Although this format also stores architecture, weights, and training configuration in a single file, it is now advertised as *"legacy"*: in Keras 3, loading an `.h5` file raises a warning.

Non-self-contained formats. Keras supports weights-only persistence through the `save_weights` and `load_weights` methods [37]. In this case, the weights are stored in a `.h5` file. However, restoring them requires instantiating the model architecture beforehand, meaning that the file alone is insufficient for full reconstruction.

2.1.2. TensorFlow. TensorFlow is an open-source Deep Learning framework developed by Google [26]. TensorFlow and Keras [25] complement each other: Keras serves as the high-level API for building and training models, while TensorFlow provides the low-level execution engine and additional abstractions. This relationship extends to model persistence. For models built with Keras layers, the recommended format is the `.keras` archive (see Section 2.1.1) [38]. By contrast, TensorFlow's native format, `SavedModel`, is designed for interoperability and deployment scenarios, particularly when working with raw TensorFlow objects or non-Python environments [27]. Excluding Keras, which has already been discussed, TensorFlow supports two persistence mechanisms: `SavedModel` and *training checkpoints* [27], [38].

Self-contained formats. The `SavedModel` format stores models in a directory containing a protocol buffer encoding a computation graph, weights (in the training checkpoints format), and optional files. Models can be restored without requiring the original model object definition.

Non-self-contained formats. Training checkpoints store only the model's weight values. As stated in the official guide, restoring a checkpoint requires the original model to be defined beforehand [28].

2.1.3. PyTorch. PyTorch is an open-source Deep Learning framework developed by the PyTorch Foundation [29]. It offers two main persistence options [30], [39]: loading the model object configuration or loading only its state dictionary (i.e., the parameters learned during training). This behavior is controlled via the `weights_only` flag. In both cases, serialization is performed using Python's pickle module (code-based format).

Self-contained formats. When `weights_only=False` and the model is not defined as a custom class, PyTorch relies on raw pickle serialization, allowing the model to be loaded without a prior class definition in the environment. However, in this case, no security restrictions are enforced to prevent arbitrary code execution, as raw pickle is known to be insecure [40].

Non-self-contained formats. `weights_only=True` requires the instantiation of a model on which to load the weights. Additionally, if the model is defined using a custom class, even when `weights_only=False`, its definition must still be available at load time, although instantiation is handled internally. Interestingly, when `weights_only=True`, PyTorch uses pickle with a restricted unpickler that limits deserialization to `torch.Tensor` objects and primitive types, and prevents dynamic imports during loading.

2.1.4. scikit-learn. scikit-learn is an open-source Python library for *traditional* ML [31]. The official documentation [32] describes several persistence approaches: (i) Python's pickle, joblib, cloudpickle modules, (ii) the Skops library [41], and (iii) conversion to the framework-agnostic ONNX format [42]. All of these are self-contained approaches, and we describe each of them in the following.

Pickle/Joblib/Cloudpickle. The traditional scikit-learn model persistence approach uses pickle, joblib, or cloudpickle. These formats serialize estimators in a code-based manner: unrestricted deserialization executes Python bytecode, posing risks of arbitrary code execution [32].

Skops. Hugging Face introduced Skops in 2022 [41], with native support for publishing models on their hub [1]. According to the official documentation [43], Skops provides `skops.io.dump()` and `skops.io.load()` as secure alternatives, designed to prevent arbitrary code execution and to reject unknown or malicious objects by default. The `.skops` archive is a ZIP file that includes, among other elements, a JSON schema (`schema.json`) describing the estimator's structure in a tree-like format, with nodes such as `MethodNode`, `TypeNode`, or `FunctionNode`. This design makes the `.skops` a data-based format, similar in spirit to Keras's `.keras` archive. The intended workflow involves calling `get_untrusted_types()` to inspect the model and identify objects that are not trusted by default. Users must then manually review these objects and explicitly allowlist them when calling `load()`; otherwise, loading fails. Therefore, security in Skops depends not only on its data-based design, but also on users' reviewing capabilities.

ONNX. scikit-learn models can be exported to the Open Neural Network Exchange (ONNX) [42] format using the `skl2onnx` converter [32]. ONNX defines a framework-agnostic format based on protocol buffers and provides a purely data-based representation, making models self-contained and portable across languages and environments. Unlike other self-contained approaches, the original estimator object (with its class structure and custom code) is not preserved. Instead, the model is reduced to a computational graph and its parameters, where the graph can only contain a limited set of predefined ML operators. ONNX serialization depends on the coverage of the chosen conversion tool, and many scikit-learn estimators remain unsupported.

2.1.5. XGBoost. XGBoost [33] is a widely used gradient boosting framework. It provides dedicated APIs for saving and loading models in either JSON or UBJSON (Universal Binary JSON) formats [44].

Self-contained formats. Unlike Deep Learning frameworks, XGBoost implements gradient boosting, which has a *closed* structure. As a result, importing the standard file format is sufficient to treat it as self-contained. However, in scenarios such as distributed or collaborative learning, where training-critical hyperparameters are not included in the JSON file (e.g., the `max_depth` parameter), the framework's authors recommend using the built-in pickle format to fully serialize the `Booster` object [34]. Alternatively, the hyperparameters must be set and provided separately.

2.2. Hub-level Sharing

An additional layer of the ecosystem is represented by model-sharing hubs. These provide infrastructures where users can publish and download pre-trained models, accelerating research and application development. Moreover, they may also incorporate further safeguards—such as content scanning, model verification, or curation policies—to protect users against threats beyond those arising from local framework-level loading. As with the frameworks, for our analysis we selected these widely used hubs: Hugging Face Hub [1], Kaggle Models [2], TensorFlow Hub [4], Keras Hub [3], and PyTorch Hub [5]. This choice is supported by usage statistics: the 2022 Kaggle ML & DS Survey [35] ranks these among the most popular ML model repositories. Table 2 summarizes their characteristics along with their security implications (discussed in Section 3).

2.2.1. Hugging Face Hub. Hugging Face Hub [1] is one of the most popular open hubs for sharing ML models. It hosts over 2.1 million models in a Git-based repository format, supporting a variety of file types. The platform implements multiple security measures [45]: every uploaded file is automatically scanned for malware using ClamAV [46], and any pickle-based model file is "pickle-scanned" to enumerate suspicious included modules. The Hub also performs secret scanning to detect accidentally leaked credentials. Furthermore, Hugging Face integrates two third-party model scanning services: Protect AI [47] and JFrog [48].

2.2.2. Kaggle Models. The Kaggle repository [2] supports both public and private model uploads, often complemented by configuration files or metrics, and is tightly integrated with Kaggle Notebooks. Many models, including those from TensorFlow Hub [4] (which fully migrated to Kaggle Models in 2023), can be loaded through high-level APIs. Security relies primarily on the isolation of Kaggle's cloud notebook environment and the use of standard model formats, as the platform does not publicly document malware or pickle scanning for user-contributed models.

2.2.3. PyTorch Hub. PyTorch Hub [5] is a built-in model-sharing mechanism for PyTorch that enables users to

TABLE 2: Hub-level ML model sharing mechanisms and security considerations, summarizing the characteristics of the analyzed model-sharing hubs and indicating their hosting model, reference format (i.e., the specific model format the hub relies on, if any), and their associated security models.

Hub	Characteristics		Security model (security posture + commentary)	
	Centralized hosting	Reference format	SO	Comments
Hugging Face Hub [1]	●	-	✔	Malware, pickle, and secret scanning; integrates third-party model scanners
Kaggle Models [2]	●	-	✖	No explicit security measures documented; relies on notebook isolation and standard formats
PyTorch Hub [5]	○	-	✖	No explicit security measures documented; arbitrary Python execution from repositories
TensorFlow Hub [4]	●	`SavedModel`	✖	No explicit security measures documented
Keras Hub [3]	●	`.keras`	✖	No explicit security measures documented

Legend: SO = Security-Oriented, ● = Supported, ○ = Not supported, ✔= Yes, ✖= No. Green = mechanisms that present security-oriented features.

load models from GitHub repositories with a single command [5]. Unlike centralized platforms, it is entirely decentralized: models are hosted on GitHub, and PyTorch Hub collects only an entry-point script (`hubconf.py`) for fetching and instantiating them. The PyTorch team maintains a list of models on the official PyTorch Hub page (via a pull-request submission process) [49], but using the `torch.hub` APIs, it is possible to load models from any public repository by URL. This flexibility means that `torch.hub.load()` downloads the target repository and executes its model-loading code. PyTorch Hub performs no automated security scanning or integrity checks.

2.2.4. TensorFlow Hub. TensorFlow Hub [4] was originally released alone as a central repository for reusable TensorFlow modules. It provides a library and hosting platform where models can be published and imported via simple APIs, such as `hub.KerasLayer`. Unlike decentralized solutions, TensorFlow Hub offers a collection of models maintained either by Google or by community contributors. Since 2023, TensorFlow Hub has been fully integrated into Kaggle Models [2], and while the API remains available, models are uploaded directly to Kaggle. By default, it adopts the `SavedModel` format without additional restrictions and makes no public mention of automated scanning.

2.2.5. Keras Hub. Keras Hub [3], introduced with Keras 3, provides a collection of ready-to-use pretrained Keras models. Unlike TensorFlow Hub, which hosts TensorFlow-specific modules, Keras Hub focuses exclusively on models designed to integrate seamlessly with the Keras API. Models can be loaded directly via `.from_preset()` and are distributed in the standardized format `.keras`. No content scanning is explicitly documented.

3. Model Sharing Security Implications (RQ1)

This section analyzes the security implications of the sharing approaches adopted by frameworks and sharing hubs described in Section 2, aiming to clarify the current state of security in ML model sharing and highlight critical issues requiring further investigation. Before the analysis, we formally define the threat model underlying our study.

3.1. Threat Model

We consider the threat posed by malicious ML model artifacts that target users loading models through popular ML frameworks. Our threat model defines the attacker's target, objectives, and capabilities. Finally, we discuss the relevance of our threat model in the real world.

Attacker's Target. Our analysis focuses on the ML model-loading pipeline executed locally on a user's machine. The system the attacker targets consists of: (i) the *user environment*, including the operating system with an installed ML framework (e.g., PyTorch [29], TensorFlow [26], Keras [25], scikit-learn [32]); (ii) the *model artifact*, a serialized pre-trained file such as `.pth`, `.keras`, `.h5`, or `.pkl`; and (iii) the *loading mechanism*, i.e., framework-provided functions such as `torch.load()` [30]. We focus on scenarios where a user loads a model obtained from an external source.

Attacker's Goal. We consider an attacker who crafts a malicious model artifact to compromise the victim's system upon loading. The primary goal is to achieve arbitrary code execution. Such attacks typically exploit vulnerabilities in the deserialization process, targeting the host system itself rather than the functionality of the model.

Attacker's Capabilities. The attacker can create, modify, and distribute ML model artifacts but has no prior access to the target's system and cannot influence the victim's environment, configuration settings, or model-loading flags. We consider two main distribution channels: (1) *public repository poisoning*, where a malicious model is uploaded to a trusted platform (e.g., Hugging Face Hub [1], Kaggle, GitHub) under the mask of a legitimate resource; and (2) *direct delivery*, where the artifact is sent to the victim via private channels such as email or cloud storage.

Real-World Relevance. Our threat model is grounded in recent large-scale measurements showing that malicious model artifacts exist in practice (e.g., 91 malicious models and several poisoned dataset scripts discovered by monitoring Hugging Face over a three-month period) [10], [16]. While these studies focus only on known vulnerabilities in model deserialization (e.g., Python's pickle unsafe deserialization [40]), they demonstrate the real-world relevance of our threat model. Furthermore, according to a recent

Hugging Face report, Protect AI [47] has scanned over 4.47 million unique model versions across 1.41 million repositories, identifying 352,000 unsafe or suspicious issues in 51,700 models as of April 2025 [50].

3.2. Security Analysis

The frameworks described in Section 2 adopt or recommend different strategies for model sharing. On top of these mechanisms, model hubs provide their own approaches for distributing pre-trained models. Here, we systematically analyze whether and how frameworks and hubs address the risks defined in the threat model, starting from the official documentation they provide. In particular, we classify *framework-level* and *hub-level* mechanisms as *security-oriented* (directly claiming or implicitly providing security properties) or *non-security-oriented*. Tables 1 and 2 summarize the classification and corresponding security models.

SA.F - Framework-level Security Analysis.

Security-Oriented Formats. Keras and scikit-learn provide formats embedding the entire model object (self-contained approach) that are explicitly presented as security-oriented in their documentation: `safe_mode=True` for Keras [22] and Skops for scikit-learn [32]. Keras's `safe_mode` flag disables *"unsafe lambda deserialization"* [22], thereby preventing execution of pickle-encoded payloads. Skops enforces a trusted type validation step [43], requiring users to manually review and approve potentially insecure objects before loading (via `get_untrusted_types()`). The developers describe this mechanism as *"secure persistence"* [43], but also explicitly caution that the library is still under active development and may contain unresolved security issues. Both Keras and Skops base their security models on avoiding pickle in favor of declarative, JSON-based formats. However, our analysis in Section 4 shows that this assumption does not always hold. In particular, although JSON itself is a data-based format, the way Keras and Skops process their JSON-based files effectively makes them behave like code-based formats—creating critical issues discussed later in the paper.

XGBoost uses a JSON/UBJSON-based format [34] that saves no executable code. To resume training, hyperparameters must be retrieved and set separately, while the architecture code is standardized and provided by the XGBoost library itself. Security is implicitly provided by the fixed architecture of the XGBoost model. Notably, the developers do not advertise this design as security-oriented. Furthermore, the documentation explicitly recommends raw pickle serialization in distributed or collaborative scenarios (where preserving hyperparameters is needed to resume training), reintroducing well-known risks of arbitrary code execution.

ONNX [42] does not explicitly make security claims in its documentation. However, scikit-learn refers to ONNX as the *"most secure solution"* for model persistence [32]. This characterization derives from ONNX's operator-based design (see Section 2.1), which inherently limits the attack surface. Its restricted expressiveness—a limited set of predefined operators, optionally extendable through external libraries—makes arbitrary code execution unlikely. Nevertheless, while ONNX is well-suited for inference (scikit-learn recommends ONNX only for that purpose [32]), it is far less flexible for training.

Non-self-contained formats such as PyTorch's `weights_only` serialization, Keras's weights-only API, and TensorFlow's training checkpoints can also be considered relatively security-oriented. Note that PyTorch explicitly recommends the `weights_only` mode in its documentation, claiming the presence of security measures that are not enforced when this mode is disabled. These approaches store only numerical parameters, making them inherently security-oriented. However, they merely shift the trust problem: model architecture code must be provided separately, and if obtained from unverified sources, it reintroduces the same arbitrary code execution risks.

Non-Security-Oriented Formats. Other frameworks rely on inherently insecure formats. Keras `safe_mode=False` or legacy HDF5, PyTorch `weights_only=False`, scikit-learn (pickle-based), and XGBoost (model + hyperparameters) all persist models using unrestricted pickles. TensorFlow's `SavedModel` format [27], despite being based on computation graphs, is explicitly described by TensorFlow developers as insecure when loading untrusted models [51].

SA.H - Hub-level Security Analysis.

Security-Oriented Hubs. Among the analyzed hubs, Hugging Face Hub [1] stands out as the only platform with active and documented security measures. These include malware scanning, pickle scanning, and secret scanning, complemented by integrations with external services such as Protect AI [47] and JFrog [48]. When a model artifact is uploaded, it is subjected to these scans, and if no issues are detected, the platform marks the model with a "Safe" label. Overall, this approach reflects Hugging Face's recognition of models as executable code and its commitment to enforcing security practices consistent with that perspective.

Non-Security-Oriented Hubs. TensorFlow Hub [4] and Keras Hub [3] do not perform systematic artifact scanning; instead, they offload security to the framework level by relying on controlled formats (`.keras`, `SavedModel`). Kaggle benefits from community curation and the isolation of its notebook environment, though unverified models can still be uploaded. PyTorch Hub provides no hub-level protections at all: it executes arbitrary Python code from repositories, leaving responsibility entirely to the user.

4. When *secure* is not secure (RQ2)

While existing evidence, both from prior scientific studies [10], [16] and from independent checks by model hubs [50], has brought attention to the relevance and scale of attacks targeting ML model loading mechanisms, these efforts have largely focused on well-known attack vectors. In particular, they have examined inherently insecure methods, such as pickle deserialization, but have not questioned the

effectiveness of mechanisms that are claimed to be secure. In this section, we go a step further: building on the observations from our security analysis in Section 3, we evaluate how closely the security narrative promoted by popular ML frameworks and hubs, claiming to secure the entire model-sharing process, matches reality.

As discussed, some approaches are deliberately *non-security-oriented*, relying on insecure formats or distribution methods. Others offer limited security but at the expense of flexibility or by shifting responsibility to users or other workflow components (e.g., requiring the model architecture to be separately obtained). A smaller subset explicitly presents itself as *security-oriented*, seeking to make model sharing more secure overall. This section focuses on hubs that provide scanning mechanisms and on frameworks that claim security combined with complete model object sharing. In particular, from a framework-level perspective, we assess the solutions highlighted in Table 1: the `safe_mode` in Keras and the *"secure persistence"* [43] of Skops for scikit-learn. From a hub-level perspective, we analyze the only hub that provides embedded scanning mechanisms, as highlighted in Table 2: the Hugging Face Hub.

Notably, our analysis uncovered six 0-day vulnerabilities (all assigned to CVEs) across Keras and Skops, each allowing arbitrary code execution during model loading. Interestingly, both frameworks base their security claims on data-based format persistence, which our findings show does not hold in practice. Moreover, our hub-level experiments demonstrated that it is feasible for an attacker to distribute exploits leveraging those vulnerabilities without being detected by the scanning tools integrated into Hugging Face.

Vulnerability Research Methodology. All vulnerabilities described below were discovered through manual reverse engineering of the latest Keras and Skops open-source codebases from GitHub, focusing on security-related functionalities.

Pre-existing CVEs. We reviewed publicly available CVEs for Keras and Skops. Prior to our work, no CVEs had been assigned to the `.keras` format or `safe_mode`, and only one to Skops. A detailed analysis is provided in Appendix A.

4.1. Keras

KV.1 - Abusing Insecure Module Resolution. We discovered that due to insufficient validation in the Keras model loading process, a carefully crafted `config.json` file inside a `.keras` model archive can specify insecure Python modules and functions to be imported and executed during loading. In particular, an attacker can build a `config.json` such that, for example, `subprocess.run` is interpreted as a model layer. Keras's loading logic performs minimal validation: if the specified class name resolves to a Python `FunctionType`—which `subprocess.run` does—no further checks are performed. Arguments can then be passed abusing the input–output relations within Keras's internal computation graph, enabling execution of commands such as `subprocess.run("/bin/sh")`. Since the model layer is instantiated at load time, the command is executed during loading. A simplified `config.json` snippet is provided in the appendices (Listing 1), while the complete PoC is available on Zenodo. Crucially, this finding demonstrates the existence of entirely different exploitation paths in Keras's loading mechanism, beyond the abuse of `Lambda` layers.

Disclosure. After coordinated disclosure, the vulnerability was assigned the identifier `CVE-2025-1550` (CVSS 7.3, CNA: Google LLC) [23]. To the best of our knowledge, this is the first CVE assigned to Keras's model loading mechanism after the introduction of `safe_mode`, and thus the first to demonstrate a weakness in this security feature. The issue was mitigated in Keras version 3.9 through the introduction of stricter validation, which restricts imports to a limited set of trusted modules, primarily within Keras itself (allowlist approach).

KV.2 - Code Reuse via `Lambda` Layers. `Lambda` layers in Keras can be used not only to serialize Python bytecode but also to reference functions from specified Python modules. We found that an attacker can achieve arbitrary code execution through *code reuse* by leveraging `Lambda` layers to abuse legitimate functions from Keras's internal modules, effectively bypassing the restriction introduced after KV.1 disclosure. As a PoC, we crafted a model that disables `safe_mode` during loading, even if it was initially enabled by the user, by invoking an internal Keras utility. Then, by relying on other internals, such as the function used to load the model (now with `safe_mode` disabled), we show how to achieve full code execution. Our PoC represents just one possible execution path; different exploits can be built using other internal functions (execution gadgets). A demonstrative snippet is provided in the appendices (Listing 2), while the complete PoC is available on Zenodo.

Disclosure. We disclosed the vulnerability through a coordinated disclosure process. Two distinct CVEs were assigned to this vulnerability. The first (`CVE-2025-8747`, CVSS 8.6, CNA: Google LLC [24]) refers to the threat of arbitrary file download through specific gadget reuse, which we used as an optional step in our complete PoC, with our report considered a contemporary independent report of that by JFrog researcher Andrey Polkovnichenko [52]. The second (`CVE-2025-9906`, CVSS 8.6, CNA: Google LLC [53]), instead, concerns code reuse to disable `safe_mode`, thereby covering the broader and more severe threat of arbitrary code execution. The fix extends the validation checks introduced after KV.1 by enforcing that any object accessed from an imported module must be an instance of `KerasSaveable`. Additionally, several internal Keras utilities that could be abused were blocklisted, including the ones we used in our PoC.

KV.3 - Silent Bypass via Legacy HDF5 Format. As noted in Section 2.1.1, Keras continues to support loading legacy models in the HDF5 format. Because of their legacy nature, some security checks introduced for the `.keras` format do not apply to HDF5 models. Specifically, there

is no mechanism to restrict the content of `Lambda` layers [22]. However, we observed that when an HDF5 model is loaded using `load_model(..., safe_mode=True)`, the `safe_mode` flag is silently ignored—without any warning or error, even though such feedback would be reasonably expected given the impossibility of enforcing this mode. Technically, the argument is never forwarded to the internal legacy loading routine and therefore has no effect. In this case, no sophisticated techniques are required for an attack, as the legacy format permits deserialization of arbitrary code through unrestricted `Lambda` layers. An attacker can then exploit this misleading behavior, leveraging the fact that users may blindly trust the presence of a flag labeled as "safe." The PoC is available on Zenodo.

Disclosure. As in the previous cases, we disclosed the issue to the Google Open Source Security Team, which serves as a coordination channel for Google-affiliated open-source projects. However, this time our issue was marked as *"Won't Fix (Infeasible)."* After a constructive discussion, Google Security clarified that they *"won't treat Keras `safe_mode` as a security boundary"* anymore and that they *"just don't think `safe_mode` is reliable enough to be a security boundary"*, further explaining that *"the panel changed its view on this issue in [KV.2 issue]"* and stating: *"Don't rely on `safe_mode` (maybe a poor name) for that level of protection."* Moreover, they also clarified that *"we don't speak for the Keras team—they might see it differently. If you want to see code changes in Keras for patching these issues, GitHub's the place to make it happen"* [54]. Following this upstream referral, we then contacted the Keras team through GitHub's private advisory. Keras acknowledged the concern and fixed the issue in Keras version 3.11.3. The fix extends `safe_mode` to the legacy file format as well: the flag is now forwarded to legacy loading, ensuring that `Lambda` layers are constrained in the same way, including the additional restrictions introduced for KV.2. The vulnerability has been assigned the identifier `CVE-2025-9905` (CVSS 7.3, CNA: Google LLC) [55].

4.2. Skops

SV.1 - Abusing `MethodNode`. In a Skops model, the `MethodNode` allows access to Python object attributes using dot notation. However, shortcomings in its design allow for traversal of the object graph of legitimate objects and access to sensitive Python internals. These can then represent powerful primitives, ultimately enabling arbitrary code execution during model loading. For example, a legitimate object may be instantiated using an `ObjectNode`, which enforces type validation and permits only trusted or explicitly allowed types. Once the object is in memory, however, an attacker can chain multiple `MethodNode` entries to traverse the object graph and access runtime structures such as `__builtins__`, which exposes dangerous functions like `eval` and `exec`. Furthermore, while the `__class__` and `__module__` fields of a `MethodNode` are validated when `get_untrusted_types()` and `load()` are called, the `func` field determines which attribute is accessed, allowing malicious attribute traversals to go unnoticed.

During our analysis, we achieved arbitrary code execution using an apparently benign type returned by `get_untrusted_types()`, such as `builtins.int`. The specific type is irrelevant, as it is only checked during validation and never used by the Skops loading logic; in practice, any string can be substituted without affecting the outcome. A representative `schema.json` snippet from the `.skops` file of our PoC is provided in the appendices (Listing 3), and the full PoC is available on Zenodo.

Disclosure. We disclosed the vulnerability to the Skops team via GitHub's private advisory and actively collaborated with the maintainers to develop and validate a mitigation. The fix was included in Skops version 0.12.0. The patch enforces that the `__module__` and `__class__` entries match those of the actual object passed to the `MethodNode`. In addition, it extends the untrusted types reported to the user by including any attribute accessed via `MethodNode`—that is, the concatenation `__module__.__class__.func`—thereby enabling human validation of the `func` entry as well. The vulnerability has been assigned the identifier `CVE-2025-54413` (CVSS 8.7, CNA: GitHub, Inc.) [56].

SV.2 - Bypassing Validation via `OperatorFuncNode`. The `OperatorFuncNode` allows invoking methods from Python's `operator` module. However, similar to SV.1, we observed a mismatch between what is validated by `get_untrusted_types()` and `load()` and what is internally used by Skops during model loading, enabling unnoticed access to operator methods. Specifically, while the concatenation of the `__module__` and `__class__` fields is validated, the `__module__` value is ignored and only `__class__` is used. In practice, if `__class__` is set to the string "some_method", the function actually invoked is `operator.some_method`, regardless of the value of `__module__` .This allows an attacker to supply a misleading, seemingly benign module path that passes validation, while the executed function is actually taken from the `operator` module. This ultimately enables code execution through methods such as `operator.call`, which invoke arbitrary targets with attacker-controlled arguments. A `schema.json` fragment showing the attack core is provided in the appendices (Listing 4), while the PoC demonstrating arbitrary code execution is available on Zenodo.

Disclosure. We disclosed the vulnerability to the Skops team via GitHub's private advisory system and collaborated with the maintainers for remediation. The issue was resolved in Skops version 0.12.0 by enforcing the `__module__` field of an `OperatorFuncNode` to be set to "operator". The identifier `CVE-2025-54412` (CVSS 8.7, CNA: GitHub, Inc.) has been assigned [57].

SV.3 - Silent Fallback to joblib in Model Card. As previously noted in Section 2.1.4, Skops is developed with integration into the Hugging Face ecosystem in mind. To

TABLE 3: Detection results of Hugging Face scanning tools for our PoCs and baselines, as presented in the interface.

Test (format)	Picklescan	ClamAV	Protect AI	JFrog	Final label
KV.1 (.keras)	not a pickle	No issue	Unsafe	No Issue	Unsafe
KV.2 (.keras)	not a pickle	No issue	Suspicious	(**)	(**)
KV.3 (HDF5) (same as M-L)	not a pickle	No issue	(*)	Unsafe	Unsafe
SV.1 (.skops)	not a pickle	No issue	No issue	not a model	Safe
SV.2 (.skops)	not a pickle	No issue	No issue	not a model	Safe
SV.3 (pickle)	not a pickle	No issue	Unsafe	Unsafe	Unsafe
B-L (HDF5)	not a pickle	No issue	Suspicious	Unsafe	Unsafe
B-L (.keras)	not a pickle	No issue	Suspicious	Unsafe	Unsafe
M-L (.keras)	not a pickle	No issue	No issue	Unsafe	Unsafe
NL (HDF5)	not a pickle	No issue	No issue	No issue	Safe
NL (.keras)	not a pickle	No issue	No issue	No issue	Safe

(*) The scanning tool did not return any results (i.e., an empty label).
(**) The scanning tool remained stuck in the "Queued" status. No final label was therefore computed. Re-uploading the model produced the same result.
Legend: B = Benign, M = Malicious, L = `Lambda`, NL = No `Lambda`.

support this, Skops provides an API for generating model cards, which serve as structured documentation for models and include fields such as description, authors, diagrams, etc. [58]. When a `Card` object is created—typically specifying the model file and, optionally, a list of trusted types—Skops internally invokes `Card.get_model()` to load the associated model. If the provided model file is in the `.skops` format (i.e., a ZIP archive), the standard Skops `load()` function is used, applying all the security checks already discussed. However, if the provided file is not a valid ZIP archive, the fallback mechanism silently switches to using `joblib.load()` to deserialize the model, without warning the user. Nevertheless, joblib does not provide the same protections as Skops and allows arbitrary code execution via pickle-based deserialization. Importantly, this behavior is based on the file's actual format—not its extension—making it especially difficult for users to detect. An example PoC is available on Zenodo.

Disclosure. We disclosed the vulnerability to the Skops team via GitHub's advisory system and proposed a fix to the maintainers, which was accepted and included in Skops version 0.13.0. The fix disallows the use of joblib unless explicitly authorized by the user through the new `allow_pickle` argument during `Card` creation. The vulnerability has been assigned the identifier `CVE-2025-54886` (CVSS 8.4, CNA: GitHub, Inc.) [59].

4.3. Hugging Face

Following the threat model described in Section 3.1, we assess whether an attacker can distribute exploits that target framework-level vulnerabilities triggered during model loading, without being detected by the scanning tools integrated into model hubs. In other words, we assess whether these scanners provide an effective additional line of defense when framework-level protections fail. To do so, we tested the exploits for vulnerabilities we identified in Keras and Skops against the scanners integrated into Hugging Face, which is the only major hub that integrates scanners. We uploaded the original PoCs for each vulnerability without obfuscation or modification. Each exploit opens `/bin/sh`. As a baseline, we also uploaded a set of additional Keras models: one containing a benign `Lambda` layer (doing nothing), one containing a *malicious* `Lambda` layer calling `/bin/sh`, and one with no `Lambda` layers. For each of these, we uploaded both the HDF5 and `.keras` versions[1]. Our experiments were performed several weeks after the CVEs were publicly disclosed, assessing the capability of scanners to detect public threats replicable by attackers.

We made all loaded artifacts (or the corresponding generation scripts) publicly available. Before uploading any malicious models, we requested and obtained explicit permission from Hugging Face. The results are shown in Table 3.

MS - Effectiveness of Hub-Integrated Model Scanners.

Picklescan. As evident from Table 3, this scanner does not detect any of our PoCs, classifying every file as "not a pickle," including the malicious pickle of KV.3. The reason lies in the extension chosen for the PoC (`.skops`). To confirm it, we uploaded the same pickle file under different names. Files with extensions such as `.h5`, `.onnx`, `.pkl`, and `.pt` were flagged as (malicious) pickle, while `.json`, `.skops`, or `.keras` were flagged as "not a pickle".

ClamAV. All model scans reported "No issue." This result is expected, as ClamAV is a general-purpose malware scanner and is not designed to detect ML framework-level threats.

Protect AI. Protect AI's Guardian [60] scanner defines a set of model-sharing threats [61]. When a model is uploaded to Hugging Face, the scanner checks the files against these definitions and generates a report that alerts users if any threats are detected. In our test, this scanner returned matches for KV.1, KV.2, and two baseline models.

For KV.1, the PoC triggered the threat "*PAIT-KERAS-301: Keras Model Custom Layer Detected at Model Run Time,*" which explicitly covers cases where non-Keras layers are included within a Keras model. This corresponds exactly to the basic PoC we uploaded, confirming the tool's ability to identify such cases. This result was somewhat expected, as following the publication of the CVE associated with KV.1, Hugging Face announced new Protect AI threat definitions, explicitly citing our CVE as an example of the type of threat detectable through those updates [50]. KV.2 was instead flagged as "suspicious" under the threat *"PAIT-KERAS-100: Keras Model Lambda Layer Can Execute Code At Load Time."* However, as demonstrated by our baseline models, this threat definition is generic and triggered by the presence of `Lambda` layers, rather than their internal operations. As a result, benign `Lambda` layers also trigger this warning, leading to false positives. This may, in turn, reduce users' perceived severity due to alert fatigue. Interestingly, the baseline containing a `Lambda` layer that invokes `/bin/sh` in a `.keras` file was not flagged (i.e., a false negative), while the equivalent model in `.h5` format (KV.3) did not produce any label. Overall, this test raises concerns regarding the accuracy of this particular threat definition.

1. It is worth noting that the baseline model with a `Lambda` layer calling `/bin/sh` in HDF5 format coincides exactly with the PoC of KV.3, since the exploit consists of using a legacy HDF5 model.

Regarding the Skops models, the pickle file for SV.3 was correctly detected, as expected for a standard (malicious) pickle file. However, neither of the `.skops` files for SV.1 or SV.2 raised any issues. This is consistent with the fact that no threats are explicitly defined for Skops in Guardian. However, it is concerning that these cases were flagged as "No issue" rather than returning a more neutral (and informative) label indicating a lack of compatibility.

JFrog. JFrog correctly classified the baseline Keras models containing a malicious `Lambda` as "Unsafe," and the non-malicious Keras models without `Lambda` layers as "No issue." It also successfully identified the malicious pickle file of SV.3. However, similar to Protect AI, it misclassified Keras models with benign `Lambda` layers as "Unsafe"—an even stronger flag than the one assigned by Protect AI—thus producing false positives. The reported details cited *"models with Lambda layers containing malicious code,"* which did not align with reality. JFrog also failed to detect our exploit for KV.1, demonstrating difficulties in identifying newer Keras threats, and provided no decision for KV.2, which remained stuck in the status "Queued" despite being uploaded to the same repository and at the same time as the others. Re-uploading KV.2 yielded the same result. Finally, both `.skops` models (SV.1 and SV.2) were labeled as "not a model". While this outcome can be explained by the lack of support for the `.skops` format, it may nevertheless be misleading for end users.

Final Label. The final label shown by Hugging Face corresponds to the most severe label assigned by its scanners. This approach allows threats such as the PoC for KV.1 (detected only by Protect AI) to be flagged as "Unsafe," thereby compensating for false negatives. On the other hand, this strategy also amplifies false positives: harmless Keras models with `Lambda` layers are escalated to "Unsafe." Concerningly, since none of the scanners support Skops models, the final label for both SV.1 and SV.2 PoCs was "Safe." This represents a clear false negative and is particularly problematic, as a reassuring label such as "Safe" was assigned despite the absence of compatible scanners.

5. Survey on User Perception (RQ3)

To assess whether the narratives promoted by certain frameworks and hubs influence ML practitioners' perceptions, we conducted a public survey. The survey was distributed via social media and direct outreach to professionals in both academic and industrial ML communities. To minimize bias, we did not disclose the cybersecurity focus of the study. This enables gathering more authentic insights into the natural concerns and mental models that participants associate with model sharing. Responses were anonymous, and no sensitive data was collected. The survey included 14 multiple-choice questions, 6 of which allowed optional open-ended answers. It was structured into three main parts, presented in the following sections. All questions, raw results, and scripts used for both statistics extraction and plotting are publicly available.

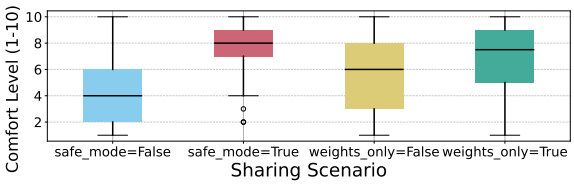

Figure 1: Distribution of user comfort levels (1–10) when loading shared models under different configurations.

Limitations. While the survey offers insights, its limited number of participants means the findings should be considered as indicative rather than representative of the broader ML community. Nevertheless, the results highlight meaningful trends in how model sharing security is perceived. To assess the robustness of these observations, we complement the analysis with statistical tests evaluating the significance of the reported outcome.

UP.1 - Demographics. A total of 62 participants completed the survey. Among them, 53 (85.5%) reported experience loading or sharing ML models. All results presented in the following analysis are restricted to these 53 participants.

Within this group, 33 (62.3%) listed ML or Artificial Intelligence (AI) as their primary area of expertise, 9 (17.0%) selected cybersecurity, and the remaining 11 (20.7%) came from related fields such as data science, software engineering, high-performance computing, or robotics. The average self-assessed expertise in ML was 3.47 on a scale from 1 (basic familiarity) to 5 (expert).

UP.2 - Perception of Model Loading Security. To assess participants' perceived security when loading shared models, we asked them to rate their comfort with four Python scripts—two using Keras and two using PyTorch. These frameworks were selected because both expose security-oriented loading-time flags explicitly: `safe_mode` in Keras and `weights_only` in PyTorch. Together, they also cover a range of persistence strategies—self-contained vs. non-self-contained, and data-based vs. code-based formats—providing valuable contrast for our analysis. For each case, participants were also asked to specify the reasons for any lack of comfort, selecting from predefined categories—such as ethical concerns, data bias, and cybersecurity risks—or by providing an open-ended response.

Keras `safe_mode`. The survey included two Keras code snippets differing only in the value of the `safe_mode` flag (`False` vs. `True`). As shown in Figure 1, average comfort levels grow from 4.5/10 with `safe_mode=False` to 7.8/10 with `safe_mode=True`, with responses shifting from widely dispersed to tightly clustered around higher scores—indicating stronger perceived security. The number of participants expressing concerns about arbitrary code execution decreased from 44 (83%) to 8 (15.1%), half of whom identified cybersecurity as their primary area of expertise.

PyTorch `weights_only`. The survey included two PyTorch snippets, each setting the `weights_only` flag to

either `False` or `True`. In the latter case, participants were informed that model definitions must be provided separately, possibly by importing shared code. Here, the shift in perceived security was smaller than in the previous case: the average comfort level grew from 5.9 to 7.0. As shown in Figure 1, the dispersed distribution of responses indicates uncertainty among participants. However, the number of participants expressing concerns about arbitrary code execution decreased more sharply, though still less than in the Keras case, from 28 (52.8%) to 13 (24.5%), even though the *more secure* option requires manually loading code for the model definition. Notably, despite unrestricted pickle use, respondents reported moderate concern in the `weights_only=False` setting.

Statistical Validation of the Observed Effects. To evaluate whether the observed differences in perceived security are statistically robust, we conduct a paired Wilcoxon signed-rank test [62]. In this context, p denotes the p-value of the test and r the corresponding effect size. The results indicate a statistically significant increase in reported comfort when `safe_mode=True` with a large effect size ($p = 3.14 \times 10^{-9}$, $|r| = 0.87$). A statistically significant comfort increase, though more moderate, was also observed when `weights_only=True` ($p = 0.0054$, $|r| = 0.42$). These results indicate that the differences in perceived security reported in Figure 1 are unlikely to arise from random variation and suggest that the presence of security-oriented mechanisms meaningfully influences user perception of model loading security.

UP.3 - Impact of Model-Sharing Hubs. We asked participants whether their comfort level would change when loading models from hubs such as Hugging Face, which integrates various security scanners. In total, 39/53 participants (73.6%) responded that the use of such hubs would *increase* their level of comfort when loading shared models.

5.1. Results Interpretation

These results highlight a clear shift in user perception driven by the presence of specific flags in the model-loading function, particularly among participants who did not identify as cybersecurity experts. Notably, the results indicate a stronger influence of `safe_mode` compared to `weights_only`. This discrepancy may stem from users perceiving the need to manually define and instantiate the model (required by `weights_only=True`) as an additional source of risk. At the same time, it may also reflect how different flag-naming choices shape users' perception of security. Finally, the results confirm that hub-level scanning features also have a significant effect on user perception.

6. Takeaways

In this section, we take a broader perspective to discuss the implications of our findings. Our discussion is enriched with suggestions for the community, directed at both ML practitioners and framework or hub maintainers, to raise awareness and enable sound security choices.

6.1. The Illusion of *Secure* File Formats

Derived from: SA.F, KV.1, KV.2, SV.1, SV.2

Data-based formats are often perceived as *more secure* than code-based formats, primarily because they do not directly serialize executable code. This belief is reinforced by the documentation and narrative of explicitly security-oriented frameworks. While this claim may seem reasonable, our analysis in Section 4 demonstrates that the file format alone does not determine the security of model sharing, and that the reality is far more complex. In particular, both Keras and Skops use JSON to represent the logical structure of executable code. Security, therefore, depends on strict validation and restrictions applied during the translation from JSON to code objects, resulting in a threat model similar to that of code-based formats. This fragility was evident in our discovered vulnerabilities (KV.1, KV.2, SV.1, and SV.2) and the subsequent patches. Indeed, these fixes restricted the excessive flexibility and addressed missing checks in how code was reconstructed from JSON. In other words, model expressiveness impacts security regardless of the serialization format used: higher expressiveness increases the attack surface. Alternative approaches exist, such as constrained formats like ONNX [42], which implicitly reduce the attack surface by limiting the set of allowed operators, albeit at the cost of flexibility (see Section 6.2).

JSON Models Are Code. To conclude this point, we report a statement from the Google Security Team during a private exchange related to one of our Keras disclosures:

"Basically, a truly `safe_mode` in Python for this isn't really possible. Loading untrusted models is like running untrusted code—models are code." [54]

Suggestions for the Community. Do not rely on file formats as a guarantee of security. What truly matters is what the format contains and how that content is processed. Sharing code objects is always inherently risky.

6.2. Block or Allow: The Security Trade-Off

Derived from: SA.F, KV.1, KV.2, SV.1, SV.2

As with software in general, designing a secure sharing mechanism requires trade-offs between flexibility, usability, and security. Across frameworks, security is often enforced via either allowlists or blocklists. While these approaches reduce the attack surface, they inevitably limit flexibility, since anything outside an allowlist or inside a blocklist is either prohibited or left to human-based fallback mechanisms.

ONNX [42], although not explicitly designed for security, restricts models to a limited set of operators, enabling interoperability across frameworks and implicitly reducing the attack surface, but at the cost of flexibility, as only certain models and training capabilities are supported.

Methods claimed to be security-oriented follow a similar pattern when patching new vulnerabilities. In Keras, the fix to KV.1 involved introducing an allowlist of Keras' modules to reconstruct the saved objects. However, as demonstrated in KV.2, this hardening could be bypassed through code reuse, requiring further restrictions and blocklisting.

PyTorch promotes its `weights_only` mode as security-oriented, as it allows only tensors, enforcing a narrow type allowlist to prevent arbitrary code execution. Keras, in contrast, while also supporting the saving of weights only, does not present this as a security measure. In both cases, the attack surface is reduced, but functionality is constrained, shifting the problem to validating the trustworthiness of the sources providing the model architecture code.

Ultimately, in some cases, allowlisting and blocklisting propagate to users themselves, who act as the last line of defense. For instance, in Skops users must manually review and approve untrusted types. However, as we demonstrate in SV.1 and SV.2, this mechanism is prone not only to human error but also to framework-level flaws, which may allow attackers to exploit user-allowlisted types while gaining capabilities for other, riskier types.

Suggestions for the Community. Learn from decades of experience in trade-offs between flexibility and security. Allowlists offer strong security, but require continuous maintenance and can affect usability. Be aware and don't rely on them alone: while they reduce the attack surface, they are not infallible, and should never be blindly trusted. Evaluate how strong limitations might offload security responsibilities to users, reintroducing risks.

6.3. The Security Cost of Slow Adoption

Derived from: KV.3, SV.3

KV.3 and SV.3 give us hints about another systemic issue: the tradeoff between legacy compatibility and security. In both cases, legacy formats using pickle deserialization (e.g., HDF5 in Keras or joblib in scikit-learn) bypass the modern security mechanisms and allow trivial code execution, even when loaded under supposedly *secure* configurations. This reflects a deeper problem in the ML ecosystem: the slow adoption of newer library versions. Backward compatibility often takes precedence to preserve reproducibility, collaboration, and portability—sometimes at the silent cost of security, as our PoCs demonstrate.

Interestingly, while the Skops fix relies on obtaining explicit user confirmation and informing users of the associated security risks, the Keras team instead chose to extend the security measures of the new format to the legacy one, demonstrating a commitment to securing legacy versions.

To give a hint of how slow the adoption of newer versions is, we analyzed downloads after the release of Keras 3.9.0 (including critical security patches). Notably, older 2.x.x versions from 2023 still saw significantly higher download counts than most newer 3.x.x releases. More data are available in Appendix B. While this does not constitute a rigorous proof, it suggests a significant inertia within the ecosystem, where users continue to rely on older versions despite the availability of known (security) improvements. Consequently, developers need to maintain support for legacy mechanisms, further complicating the design of robust and secure sharing APIs.

Suggestions for the Community. While recognizing the complexity of the problem, developers should, whenever possible, require insecure legacy options to be explicitly enabled by users and provide clear warnings about the associated risks. Users, in turn, should maintain heightened skepticism toward legacy formats, as frameworks often prioritize compatibility over security.

6.4. Model Scanning as Malware Analysis

Derived From: SA.H, MS

Automatic model scanners integrated into model hubs, while providing additional protection and being potentially useful, inherit well-known limitations of traditional signature-based malware detectors [63]. Moreover, they often adopt a framework-centric design, supporting only specific frameworks and formats, and frequently duplicate safeguards already implemented within the frameworks themselves. For instance, Protect AI [47] defines framework- and format-specific threats and checks models against them [61], restricting detection to already formalized weaknesses.

Our evaluation of Hugging Face model scanners (MS), in addition to finding false positives and negatives (see evaluation results for details), shows that Skops, despite being promoted by Hugging Face as a secure persistence format, is unsupported by any integrated scanner. This highlights the fragmented and incomplete nature of the current landscape of model scanners integrated into model hubs. Finally, the labels produced by these scanners can be misleading when no match is found. For instance, if a format is not supported, JFrog [48] returns "not a model" and Protect AI reports "No issue". Such outputs, especially if they come from a limited analysis, can create a false sense of security, such as in traditional antivirus software [64].

Suggestions for the Community. Inspired by the malware analysis domain, scanners should be treated only as a first line of defense. A possible direction is the adoption of behavioral analysis to move beyond static checks. Finally, hubs should promote greater transparency in report labels.

6.5. Trusting "Safe": A Risk in Itself

Derived from: SA.F, SA.H, KV.1, KV.2, KV.3, SV.1, SV.2, SV.3, UP.1, UP.2, UP.3, MS

What becomes evident from our analysis is that a complete and truly secure solution for model sharing does not exist. Every approach, whether from hubs or frameworks, requires compromises or shifts part of the responsibility

elsewhere. This is, unfortunately, a well-known reality in computer science, and model sharing is no exception.

However, this reality often clashes with the prevailing narrative and common beliefs. Labeling a mode or a model (after scanning) as "safe" or "secure" is rarely consistent with the actual risks involved. Instead, we should refer to these as approaches that *attempt* to provide security hardening against known exploitation paths, establish mechanisms for trusted types, or detect potential (known) threats in models prior to download. However, such efforts do not inherently make these approaches "secure" or even "safe".

While we acknowledge the complexity in designing APIs accessible to users with diverse backgrounds, such as those in the ML community, we believe there is a need for clear and transparent communication. Users often lack either the knowledge or the willingness to critically assess labels like "secure" or "safe." Such terminology is a significant oversimplification that fails to reflect the complex (and less optimistic) reality.

This kind of messaging has real consequences for users. As shown in our survey (UP.2 and UP.3), users exhibit increased security confidence when features like the `safe_mode` option are enabled or when hubs advertise built-in security scanning. While choosing methods with some degree of security hardening is certainly a positive step, it is by no means sufficient to ensure *true* security. A critical observation is that, with `safe_mode` activated, more than 90% of survey participants who did not choose cybersecurity as their field of expertise—but who had prior experience in model sharing—did not consider arbitrary code execution to be a concern. This misplaced trust is, in itself, a serious security issue that requires a coordinated community effort to be addressed.

Suggestions for the Community. Oversimplified security labels can create misplaced trust. When simplifications are necessary, present them with care to align with the actual security guarantees.

7. Related Works

Being an established research area, *generic software* supply chain security has been extensively studied, with several works [6], [7], [8] identifying and systematizing its risks. While some insights from these studies can be extended to the model-sharing context, recent research has gone a step further by specifically examining supply chain issues in this context. In particular, Meiklejohn et al. [12] framed ML models as supply chain artifacts and proposed cryptographic mitigations for the identified threats, whereas Wang et al. [11] mapped vulnerabilities across the Large Language Model (LLM) supply chain, noting that only 56% of them currently have available fixes.

Building on the observation that model hubs have become central to model distribution and therefore represent a potential vector for supply chain attacks, Jiang et al. conducted a series of empirical studies examining the security of model-sharing platforms. In their initial work, they compared model hubs to the *generic software* supply chain, categorizing them based on their access model (open or gated) and identifying the associated risks [9]. Subsequent research explored typosquatting and impersonation threats [17] and analyzed model reuse practices on Hugging Face [18]. Across these studies, Jiang et al. found insufficient provenance verification, weak dependency management, and inadequate documentation, ultimately concluding that trust in hub-shared models is often misplaced. Extending the understanding of risks and practices within model hubs, Jones et al. [19] analyzed the Hugging Face ecosystem, uncovering high model turnover rates, a correlation between popularity and documentation quality, and challenges in model management and reproducibility.

In parallel, other researchers have explored specific attack vectors. In particular, Casey et al. [10] and Zhao et al. [16] demonstrated the prevalence of insecure serialization methods (e.g., pickle) in Hugging Face models, which expose users to arbitrary code execution. Building on this, Zhao et al. conducted a large-scale measurement study of malicious code poisoning on Hugging Face, discovering multiple infected models and dataset scripts.

Focusing more on the latter threat of code poisoning, Hua et al. [13] introduced MalModel, which embeds executable payloads directly into deep learning model weights, demonstrating the feasibility of concealing malware within models. Similarly, EvilModel [14] and EvilModel 2.0 [15] showed that entire malware binaries can be hidden within model parameters without degrading performance.

From a framework-level perspective, Zhu et al. [20] introduced the TensorAbuse attack, which exploits legitimate TensorFlow APIs to perform unintended operations (e.g., file access and network messaging) during model inference. This work highlights a gap in the state of the art on the security of model sharing. This gap was further underscored by Cyrus Parzian, who presented a talk at DEF CON 33 on the risks of model sharing, using as examples the security concerns associated with pickle deserialization and ONNX models shared in `.exe` format [21].

Gaps in the State of the Art. Prior work on the security of model sharing has primarily examined supply chain risks, malicious code injection, and abuses of specific TensorFlow APIs. However, no study has systematically evaluated the security of model-sharing approaches across different ML frameworks and hubs, nor critically questioned the actual guarantees of sharing mechanisms that are claimed to be *secure*. Our paper closes this gap. We assess the security posture of widely used frameworks, uncover 0-day vulnerabilities (subsequently disclosed and assigned CVEs) in mechanisms advertised as secure but designed to support self-contained model artifacts, and examine how security narratives shape user perception. At the hub level, while prior work has focused mainly on supply-chain–related risks, we empirically assess the effectiveness of hub-integrated scanners and labeling practices, questioning their effectiveness and how they influence users' sense of security.

8. Future Directions

In line with our goal of deepening understanding of the security challenges in loading ML models, we highlight several potential directions for future research.

Beyond the framework and hub levels examined in this work, a third dimension may involve third-party libraries not included in official framework documentation and thus excluded from our selection criteria. The popularity of some (e.g., safetensors [65]) makes it worthwhile to examine how they align with our identified categories and what unique implications they introduce. Additionally, it would also be interesting to explore how users approach these libraries, specifically whether they apply a different level of skepticism compared to *official* methods.

LLM-specific formats (e.g., GGUF, GPTQ, AWQ) introduce additional challenges. Indeed, the prohibitive size of these models shifts users' focus from training to inference, although even inference is often infeasible on local machines. Moreover, pre-trained models are typically released only by a few major vendors that have the computational resources to train them. These factors not only redirect the focus of sharing solutions toward optimized formats designed for inference, which are not required to be well-suited for architectural modifications (thus making restricted solutions less limiting), but may also alter the overall threat model due to the different distribution of actors involved.

Automatic model scanners also represent a broad area for future research. While we discussed their limitations, future work could systematically assess their performance at scale, particularly against diverse adversarial techniques. Inspired by advances in malware detection, additional approaches (e.g., dynamic analysis) could be explored.

9. Conclusions

In this work, we evaluated the security posture of Machine Learning model sharing across frameworks and hubs. We found that protection is inconsistent: many mechanisms provide no safeguards, while others shift responsibility to users or impose strong restrictions on flexibility. Even those promoted as *secure* fail to reliably prevent exploitation. By uncovering 0-day vulnerabilities and analyzing user perceptions of these mechanisms, we exposed a critical gap between the security narrative and reality. Our takeaways show that there is no straightforward silver bullet. Database formats and model scanners integrated into model hubs fall short of ensuring actual security, and framework or hub naming choices further compromise user awareness. Security inevitably involves trade-offs: reducing risk often limits usability, while support for legacy formats silently reintroduces old vulnerabilities. Automatic scanning can help, but its coverage is uneven, and results are sometimes misleading. Above all, shared models must be treated as code, and loading untrusted artifacts carries the same risks as executing untrusted software.

Acknowledgments

This work was partially supported by the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. The authors would like to thank the framework and hub maintainers and developers, and the security teams involved in the coordinated disclosure process for their responsiveness, constructive engagement, and commitment to improving the security of the ML ecosystem.

References

[1] Hugging Face Inc., "Hugging Face Hub Documentation," https://huggingface.co/docs/hub/index, 2025, accessed: 2025-08-05.

[2] Kaggle, Inc., "Kaggle Models," https://www.kaggle.com/models, 2025, accessed: 2025-08-14.

[3] M. Watson, F. Chollet, D. Sreepathihalli, S. Saadat, R. Sampath, G. Rasskin, S. Zhu, V. Singh, L. Wood, Z. Tan, I. Stenbit, C. Qian, J. Bischof *et al.*, "KerasHub," https://github.com/keras-team/keras-hub, 2024.

[4] Google Research, Brain Team, "TensorFlow Hub: Reusable Machine Learning Modules," https://www.tensorflow.org/hub, 2025, accessed: 2025-08-14.

[5] PyTorch Foundation, "PyTorch Hub," https://pytorch.org/hub/, 2025, accessed: 2025-08-14.

[6] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[7] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Cham: Springer International Publishing, 2020, pp. 23–43.

[8] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1509–1526. [Online]. Available: https://doi.org/10.1109/SP46215.2023.10179304

[9] W. Jiang, N. Synovic, R. Sethi, A. Indarapu, M. Hyatt, T. R. Schorlemmer, G. K. Thiruvathukal, and J. C. Davis, "An Empirical Study of Artifacts and Security Risks in the Pre-trained Model Supply Chain," in *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, SCORED2022, Los Angeles, CA, USA, 7 November 2022*, S. Torres-Arias, M. S. Melara, and L. Simon, Eds. ACM, 2022, pp. 105–114. [Online]. Available: https://doi.org/10.1145/3560835.3564547

[10] B. Casey, J. C. S. Santos, and M. Mirakhorli, "A Large-Scale Exploit Instrumentation Study of AI/ML Supply Chain Attacks in Hugging Face Models," *arXiv preprint*, vol. abs/2410.04490, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.04490

[11] S. Wang, Y. Zhao, Z. Liu, Q. Zou, and H. Wang, "SoK: Understanding Vulnerabilities in the Large Language Model Supply Chain," *CoRR*, 2025.

[12] S. Meiklejohn, H. Blauzvern, M. Maruseac, S. Schrock, L. Simon, and I. Shumailov, "Position: Machine learning models have a supply chain problem," in *Forty-second International Conference on Machine Learning Position Paper Track*, 2025. [Online]. Available: https://openreview.net/forum?id=zfohnbkMu0

[13] J. Hua, K. Wang, M. Wang, G. Bai, X. Luo, and H. Wang, "MalModel: Hiding Malicious Payload in Mobile Deep Learning Models with Black-box Backdoor Attack," *arXiv preprint*, vol. abs/2401.02659, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2401.02659

[14] Z. Wang, C. Liu, and X. Cui, "EvilModel: Hiding Malware Inside of Neural Network Models," in *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5-8, 2021*. IEEE, 2021, pp. 1–7. [Online]. Available: https://doi.org/10.1109/ISCC53001.2021.9631425

[15] Z. Wang, C. Liu, X. Cui, J. Yin, and X. Wang, "EvilModel 2.0: Bringing Neural Network Models into Malware Attacks," *Computers & Security*, vol. 120, p. 102807, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404822002012

[16] 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, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 2087–2098. [Online]. Available: https://doi.org/10.1145/3691620.3695271

[17] W. Jiang, M. Kim, C. Cheung, H. Kim, G. K. Thiruvathukal, and J. C. Davis, ""I see models being a whole other thing": an empirical study of pre-trained model naming conventions and a tool for enhancing naming consistency," *Empirical Software Engineering*, vol. 30, no. 6, p. 155, 2025. [Online]. Available: https://doi.org/10.1007/s10664-025-10711-4

[18] W. Jiang, N. Synovic, M. Hyatt, T. R. Schorlemmer, R. Sethi, Y. Lu, G. K. Thiruvathukal, and J. C. Davis, "An Empirical Study of Pre-Trained Model Reuse in the Hugging Face Deep Learning Model Registry," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2463–2475. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00206

[19] J. Jones, W. Jiang, N. Synovic, G. K. Thiruvathukal, and J. C. Davis, "What do we know about Hugging Face? A systematic literature review and quantitative validation of qualitative claims," in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*, X. Franch, M. Daneva, S. Martínez-Fernández, and L. Quaranta, Eds. ACM, 2024, pp. 13–24. [Online]. Available: https://doi.org/10.1145/3674805.3686665

[20] R. Zhu, G. Chen, W. Shen, X. Xie, and R. Chang, "My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs," in *IEEE Symposium on Security and Privacy, SP 2025, San Francisco, CA, USA, May 12-15, 2025*, M. Blanton, W. Enck, and C. Nita-Rotaru, Eds. IEEE, 2025, pp. 486–503. [Online]. Available: https://doi.org/10.1109/SP61157.2025.00012

[21] C. Parzian, "Loading Models, Launching Shells: Abusing AI File Formats for Code Execution," Presentation at the DEF CON 33 Hacking Conference - https://media.defcon.org/DEF%20CON%2033/DEF%20CON%2033%20presentations/Cyrus%20Parzian%20-%20Loading%20Models%2C%20Launching%20Shells%20Abusing%20AI%20File%20Formats%20for%20Code%20Execution.pdf, 2025, accessed: 2025-08-21.

[22] Keras developers, "Whole model saving & loading - Keras," https://keras.io/api/models/model_saving_apis/model_saving_and_loading/, 2025, accessed: 2025-07-30.

[23] CVE Program, "CVE-2025-1550," https://www.cve.org/CVERecord?id=CVE-2025-1550, 2025, accessed: 2025-07-30.

[24] ——, "CVE-2025-8747," https://www.cve.org/CVERecord?id=CVE-2025-8747, 2025, accessed: 2025-08-17.

[25] F. Chollet *et al.*, "Keras," 2015, https://keras.io.

[26] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: a system for Large-Scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[27] TensorFlow Developers, "SavedModel format guide — TensorFlow documentation," https://www.tensorflow.org/guide/saved_model, 2024, accessed: 2025-08-17.

[28] T. Developers, "Training checkpoints — TensorFlow documentation," https://www.tensorflow.org/guide/checkpoint, 2024, accessed: 2025-08-17.

[29] PyTorch Foundation, "PyTorch Foundation," https://pytorch.org/foundation/, 2025, accessed: 2025-07-22.

[30] PyTorch developers, "Save and Load the Model — PyTorch Tutorials 2.7.0+cu126 documentation," https://docs.pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html, 2025, accessed: 2025-07-23.

[31] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[32] scikit-learn developers, "10. Model persistence — scikit-learn 1.7.1 documentation," https://scikit-learn.org/stable/model_persistence.html, 2025, accessed: 2025-07-30.

[33] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen, R. Mitchell, I. Cano, T. Zhou *et al.*, "Xgboost: extreme gradient boosting," *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[34] XGBoost developers, "Introduction to Model IO — XGBoost Tutorials 3.1.0-dev documentation," https://xgboost.readthedocs.io/en/latest/tutorials/saving_model.html, 2025, accessed: 2025-08-12.

[35] P. Mooney, "2022 Kaggle Machine Learning & Data Science Survey," https://kaggle.com/competitions/kaggle-survey-2022, 2022, kaggle.

[36] JetBrains, "Python Developers Survey 2024," https://lp.jetbrains.com/python-developers-survey-2024/, 2024, accessed: 2025-08-13.

[37] Keras developers, "Weights-only saving & loading - Keras," https://keras.io/api/models/model_saving_apis/weights_saving_and_loading/, 2025, accessed: 2025-08-04.

[38] TensorFlow Developers, "Save and load Keras models — TensorFlow documentation," https://www.tensorflow.org/tutorials/keras/save_and_load, 2024, accessed: 2025-08-17.

[39] PyTorch developers, "Serialization semantics — PyTorch 2.7 documentation," https://docs.pytorch.org/docs/2.7/notes/serialization.html, 2025, accessed: 2025-07-23.

[40] Python Software Foundation, "pickle — Python object serialization," https://docs.python.org/3/library/pickle.html, 2025, accessed: 2025-08-18.

[41] A. Jalali, B. Bossan, and Merve, "Introducing Skops," https://huggingface.co/blog/skops, August 2022, accessed: 2025-07-30.

[42] ONNX developers, "ONNX," https://onnx.ai/, 2025, version: 1.20.0.

[43] skops developers, "Secure persistence with skops — skops 0.12 documentation," https://skops.readthedocs.io/en/stable/persistence.html, 2025, accessed: 2025-07-30.

[44] XGBoost developers, "Python API Reference — xgboost 3.0.3 documentation," https://xgboost.readthedocs.io/en/stable/python/python_api.html, 2025, accessed: 2025-08-05.

[45] Hugging Face Inc., "Security — Hugging Face Hub Documentation," https://huggingface.co/docs/hub/security, 2025, accessed: 2025-08-05.

[46] Cisco Talos (ClamAV Team), "ClamAV: Open-Source Antivirus Toolkit," https://docs.clamav.net/, 2025, accessed: 2025-08-14.

[47] Protect AI, "Protect AI — The Platform for AI Security," https://protectai.com/, 2025, accessed: 2025-08-14.

[48] JFrog Ltd., "Software Supply Chain Solutions for DevOps and Security — JFrog," https://jfrog.com/, accessed: 2025-08-22.

[49] PyTorch Foundation, "PyTorch Hub — Documentation," https://docs.pytorch.org/docs/stable/hub.html, 2025, accessed: 2025-08-14.

[50] S. Morgan, "4M Models Scanned: Protect AI + Hugging Face 6 Months In," https://huggingface.co/blog/pai-6-month, April 2025, accessed: 2025-08-20.

[51] TensorFlow Security Team, "TensorFlow Security Policy," https://github.com/tensorflow/tensorflow/blob/master/SECURITY.md, 2025, accessed: 2025-08-18.

[52] A. Polkovnichenko, "Is TensorFlow Keras "Safe Mode" Actually Safe? Bypassing safe_mode Mitigation to Achieve Arbitrary Code Execution," https://jfrog.com/blog/keras-safe_mode-bypass-vulnerability/, March 2025, accessed: 2025-08-17.

[53] CVE Program, "CVE-2025-9906," https://www.cve.org/CVERecord?id=CVE-2025-9906, 2025, accessed: 2025-09-19.

[54] Google Security Team, "Private Communication," 2025, regarding the disclosure of Keras vulnerabilities.

[55] CVE Program, "CVE-2025-9905," https://www.cve.org/CVERecord?id=CVE-2025-9905, 2025, accessed: 2025-09-19.

[56] ——, "CVE-2025-54413," https://www.cve.org/CVERecord?id=CVE-2025-54413, 2025, accessed: 2025-07-30.

[57] ——, "CVE-2025-54412," https://www.cve.org/CVERecord?id=CVE-2025-54412, 2025, accessed: 2025-07-30.

[58] skops developers, "Model Cards for scikit-learn — skops 0.11 documentation," https://skops.readthedocs.io/en/stable/model_card.html, 2025, accessed: 2025-07-30.

[59] CVE Program, "CVE-2025-54886," https://www.cve.org/CVERecord?id=CVE-2025-54886, 2025, accessed: 2025-08-17.

[60] Protect AI, "Guardian — AI Model Security with Zero Compromises," https://protectai.com/guardian, 2025, accessed: 2025-08-25.

[61] ProtectAI, "Understanding Model Threats," https://protectai.com/insights/knowledge-base/deserialization-threats/PAIT-ARV-100, 2025, accessed: 2025-08-22.

[62] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[63] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Human-centric Computing and Information Sciences*, vol. 8, no. 1, pp. 1–22, 2018.

[64] E. Lau and Z. Peterson, "A Research Framework and Initial Study of Browser Security for the Visually Impaired," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4679–4696. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/lau

[65] N. Narsil, L. Liu, L. Tunstall, E. Beeching, N. Lambert, and C. Delangue, "safetensors," 11 2022. [Online]. Available: https://github.com/huggingface/safetensors

[66] CVE Program, "CVE-2024-3660," https://www.cve.org/CVERecord?id=CVE-2024-3660, 2024, accessed: 2025-08-17.

[67] J. Havrilla, A. Householder, A. Kompanek, and B. Koo, "VU#253266 - Keras 2 Lambda Layers Allow Arbitrary Code Injection in TensorFlow Models," https://kb.cert.org/vuls/id/253266, April 2024, last Revised: 2024-04-18, Accessed: 2025-08-17.

[68] CVE Program, "CVE-2021-37678," https://www.cve.org/CVERecord?id=CVE-2021-37678, 2021, accessed: 2025-08-19.

[69] François Chollet, "Disallow pickle loading in npz files," https://github.com/keras-team/keras/commit/57c94f305c0b0347ed02b11535623a8b375eee5f, January 2025, gitHub commit. Accessed: 2025-08-17.

[70] huntr, "Malicious Keras Model Leads to RCE," https://huntr.com/bounties/a3ea601c-f904-4e06-a03e-deb9ff2aa8be, February 2024, accessed: 2025-08-17.

[71] CVE Program, "CVE-2024-37065," https://www.cve.org/CVERecord?id=CVE-2024-37065, 2024, accessed: 2025-08-17.

[72] K. Schulz, "Skops Vulnerability Report," https://hiddenlayer.com/sai-security-advisory/2024-06-skops, June 2024, accessed: 2025-08-17.

Ethics Considerations

This research systematizes and empirically evaluates risks associated with loading ML models across widely used frameworks and hubs. The stakeholders for this research include ML practitioners and researchers who load pre-trained models; framework developers and maintainers; model hubs and package repositories; and security researchers. End users and society at large are only indirectly impacted.

Each vulnerability listed in this work was disclosed and discussed with the relevant maintainers in accordance with their disclosure guidelines, and we collaborated with them to design and validate effective mitigations. Misuse potentials have been mitigated through coordinated disclosure, including delaying the public release of any potentially exploit-enabling details until a timeline coordinated with the relevant maintainers and security teams.

For experiments involving the Hugging Face Hub, we obtained explicit permission to upload research models for security testing (*"if this is for security and research purposes, we grant you permission to upload your models. Please let us know if you found anything interesting."*). For the survey we conducted, participation has been totally voluntary by design, and participants were informed appropriately about the intended use of the results. Sensitive data risks related to our survey were minimized by conducting it anonymously, avoiding the collection of personally identifiable information, and adhering to applicable laws, platform terms, and community norms.

To ensure transparency, we privately shared the complete preprint version of this paper with all vendors affected by the discovered vulnerabilities or directly examined in our study, so that they were informed before any broader dissemination. This step extended beyond the vulnerabilities and findings already disclosed responsibly and involved Google, Keras, Hugging Face, and Skops. We engaged in constructive discussions when needed or upon request, while maintaining full independence in the research process.

LLM usage considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

Appendix A. Prior CVEs on Keras and Skops

We review the CVEs disclosed prior to our work concerning either Keras or Skops model-sharing methods. Our search included both the official CVE database (https://cve.org), using keywords such as "keras" and "skops", and the GitHub Security Advisory pages of the respective projects. No filters were applied to the publication time frame.

Before the publication of the first CVE we identified (KV.1) for Keras, the most recent CVE related to its model persistence was `CVE-2024-3660` [66]. This vulnerability enabled arbitrary code execution during model loading

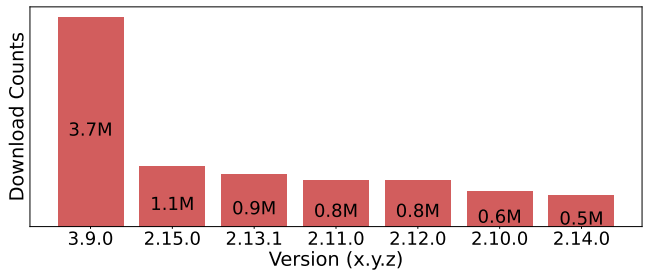

Figure 2: Download statistics of Keras versions between March 5, 2025, and March 26, 2025 (only versions with >500k downloads).

in versions of Keras prior to 2.13, which was released in March 2023. The root cause was the absence of the `safe_mode` flag (introduced only in Keras 2.13) and the unrestricted nature of the `Lambda` layers in the default model formats used at the time (HDF5), which allowed deserialization of arbitrary Python code [67]. The only other CVE related to model loading published before our work was `CVE-2021-37678` [68], which affected a YAML-based format that is no longer applicable, as it was deprecated following that disclosure. Notably, no CVEs had been assigned to the `.keras` format or to `safe_mode`, and Keras's GitHub Security Advisory page listed no advisories.

Interestingly, analysis of Keras's changelog and commit history reveals silent fixes for security-relevant issues involving `safe_mode` and not accompanied by advisories or CVE assignments. One such case is commit `57c94f3` [69] from January 2025, which addresses the insecure use of `numpy.load` with `allow_pickle=True` when loading weights from `.npz` files—a rarely used but supported alternative to HDF5. Although this issue was reported earlier on bug bounty platforms (e.g., Huntr) [70] in February 2024, it appears to have been rejected at the time.

Similarly, for Skops, we found only one prior CVE: `CVE-2024-37065` [71]. This vulnerability allowed arbitrary code execution when using the now-deprecated `trusted=True` flag. Importantly, this setting was used internally within parts of the Skops codebase, such as the update CLI tool, potentially exposing users even if they did not explicitly enable it [72]. Beyond this, we found no other vulnerabilities or advisories disclosed publicly.

Appendix B. Legacy Version Adoption Rate in Keras

We analyzed PyPI download statistics for Keras over time, following the release of version 3.9.0, which included critical security fixes. While this is not intended to be an exhaustive study, which would be outside the scope of this work, our intent is to give a hint at the adoption rate of newer versions. The statistics were collected from Google Cloud Public Datasets and queried using BigQuery. We made the queries, raw data, and scripts used to generate the plot publicly available.

```
{
   "module": "subprocess",
   "class_name": "run",
   "inbound_nodes": [
      {
         "args":[
            "/bin/sh"
         ],
         "kwargs":{

         }
      }
   ]
}
```

Listing 1: Simplified malicious `config.json` snippet for KV.1. Keras interprets `subprocess.run` as a model layer. Arguments are passed via `inbound_nodes`, which define input–output relations within Keras's model computation graph.

```
{
   "module":"keras.layers",
   "class_name": "Lambda",
   "config":{
      "name": "set_global_state",
      "function":{
         "module": "keras.src.backend.common.glob
         ↪  al_state",
         "class_name": "function",
         "config": "set_global_attribute",
         "registered_name": "function"
      },
      "arguments":{
         "value": false
      }
   },
   "name": "set_global_state",
   "inbound_nodes":[
      {
         "args":[
         ],
         "kwargs":{
            "inputs": "safe_mode_saving"
         }
      }
   ]
}
```

Listing 2: Partial malicious `config.json` snippet for KV.2. A `Lambda` layer is abused to disable safe mode via `set_global_attribute("safe_mode_saving", value=False)`. Arguments are passed through both the top-level `inbound_nodes` key and the `arguments` key within the `Lambda` layer configuration.

Figure 2 presents the most downloaded Keras versions between March 5, 2025 (the release date of version 3.9.0), and March 26, 2025 (the day before the release of version 3.9.1). This represents the most favorable time window for the adoption of version 3.9.0, which was, as expected, the most downloaded release. However, surprisingly, it was followed by two versions from 2023: 2.15.0 (1.1 million downloads) and 2.13.1 (900,000 downloads). No other version in the 3.x.x series received more than 500,000 downloads during this period and is therefore not shown in the plot.

```
{
   "__class__": "int",
   "__module__": "builtins",
   "__loader__": "MethodNode",
   "content":{
      "obj":{
         "__class__": "int",
         "__module__": "builtins",
         "__loader__": "MethodNode",
         "content":{
            "obj":{
               "__class__": "QuadraticDiscriminan
               ↪  tAnalysis",
               "__module__": "sklearn.discriminan
               ↪  t_analysis",
               "__loader__": "ObjectNode",
               "__id__": 1
            },
            "func": "decision_function"
         }
      },
      "func": "__builtins__"
   }
}
```

Listing 3: Malicious `schema.json` snippet for SV.1. `QuadraticDiscriminantAnalysis` from Scikit-learn (trusted by default) is instantiated using an `ObjectNode`. Then, a first `MethodNode` accesses its `decision_function` method and a second `MethodNode` retrieves the `__builtins__` dictionary, bypassing Skops' checks.

```
{
   "__class__": "call",
   "__module__": "sklearn.SGDRegressor",
   "__loader__": "OperatorFuncNode"
}
```

Listing 4: Malicious `schema.json` snippet for SV.2. The validated type string is `sklearn.SGDRegressor.call`, which may appear benign and related to the target model, but what is actually invoked is `operator.call`.

Appendix C. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

The paper investigates supply-chain attacks targeting ML model-loading workflows. It considers a scenario in which an adversary maliciously manipulates a model to trigger arbitrary code execution when it is loaded by a victim. The paper surveys security mechanisms for model loading in popular ML frameworks and model hubs, conducts a manual vulnerability assessment of security-oriented features, and evaluates user perceptions of model-loading safety. The analysis uncovers multiple vulnerabilities in Keras's safe_mode and Hugging Face's model scanning mechanisms.

C.2. Scientific Contributions

- Identifies an Impactful Vulnerability.
- Provides a Valuable Step Forward in an Established Field.

C.3. Reasons for Acceptance

1) The paper provides a systematic survey that highlights the fragmentation and lack of standardization in current model-sharing practices.
2) It identifies multiple exploits in secure loading modes, raising serious concerns about the effectiveness of existing safeguards in model-sharing infrastructures.