Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries

Nima Shiri Harzevili*, Jiho Shin*, Junjie Wang†, Song Wang*, Nachiappan Nagappan‡
* Lassonde School of Engineering, York University, Toronto, Canada
† Institute of Software Chinese Academy of Sciences, Beijing, China
‡ IIIT Delhi, New Delhi, India
{nshiri, jihoshin, wangsong}@yorku.ca, junjie@iscas.ac.cn, nnagappan@acm.org

***Abstract*—The application of machine learning (ML) libraries has tremendously increased in many domains, including autonomous driving systems, medical, and critical industries. Vulnerabilities of such libraries could result in irreparable consequences. However, the characteristics of software security vulnerabilities have not been well studied. In this paper, to bridge this gap, we take the first step toward characterizing and understanding the security vulnerabilities of seven well-known ML libraries, including TensorFlow, PyTorch, Scikit-learn, Mlpack, Pandas, Numpy, and Scipy. To do so, we collected 683 security vulnerabilities to explore four major factors: 1) vulnerability types, 2) root causes, 3) symptoms, and 4) fixing patterns of security vulnerabilities in the studied ML libraries. The findings of this study can help developers and researchers understand the characteristics of security vulnerabilities across the studied ML libraries.**

***Index Terms*—Security vulnerability, machine learning libraries, empirical study**

I. Introduction

Nowadays, machine learning (ML) libraries have been frequently used in a wide variety of domains including but not limited to image classification [1], [2], big data analysis [3], pattern recognition [4], self-driving [5]–[7] and Natural Language Processing [8]–[10]. These ML libraries can be vulnerable to many attacks [11], and failures to detect the vulnerabilities in these libraries could cause catastrophic outcomes, such as car accidents [12].

In the past years, there have been multiple research studies to characterize ML API bugs which are in various stages of the ML pipeline, including data preprocessing, feature engineering, model training, and inference [13]–[16], or implementation bugs which are in the code of the ML libraries, such as bugs in the core algorithms or components of the library [16]–[20]. For example, Zhang et al. [15] studied the bugs and issues that developers encounter when working with TensorFlow, a popular open-source Deep Learning (DL) library developed by Google. Islam et al. [20] conducted the first study on characterizing API usage bugs of five DL libraries, including Caffe, Keras, TensorFlow, Theano, and Torch. They provided a classification for bug types, root causes, impact, and the DL development stage where bugs occur. Despite these efforts, the characteristics of software security vulnerabilities in ML libraries have not been well studied, which leaves unanswered the more directly relevant questions: *What kinds of security vulnerabilities are found in the studied ML libraries*? *What are the root causes of security vulnerabilities in the studied ML libraries*? *What symptoms do these security vulnerabilities have*? and *Are there any fixing patterns for resolving these security vulnerabilities*?

Understanding such characteristics of security vulnerabilities in the studied ML libraries has the potential to foster the development of secure and reliable ML platforms. To fill the above research gap, we take the first step towards characterizing and understanding security vulnerabilities in ML libraries. More specifically, we conduct the first comprehensive study to explore four significant factors: 1) vulnerability types, 2) root causes, 3) symptoms and 4) fixing patterns of security vulnerabilities in seven well-known ML libraries including TensorFlow [21], PyTorch [22], scikit-learn [23], Mlpack [24], Pandas [25], Numpy [26], and Scipy [27]. In our study, we considered all available commits in the default branch of the Github repositories of the studied ML libraries when we conduct this study on Sept. 1st, 2021, to collect software security vulnerabilities. We first searched commit titles and messages with vulnerability-related keywords (details are in Section II-A) to identify vulnerability-fixing commits. As a result, more than 5K commits are collected. We then manually checked each commit collected in the first step and identify and characterize vulnerabilities from them by following systematic processes (details are in Section II-B). In total, we obtained 683 unique security vulnerabilities from the studied seven ML libraries. In this paper, we are to address the following research questions:

- **RQ1**: What types of vulnerabilities exist in the studied ML libraries?
- **RQ2**: What are the root causes for vulnerabilities in the studied ML libraries?
- **RQ3**: What are the symptoms of vulnerabilities in the studied ML libraries?
- **RQ4**: What are the fixing patterns for vulnerabilities in the studied ML libraries?

This paper makes the following contributions:

- To the best of our knowledge, we conduct the first empirical study to characterize and understand software security vulnerabilities in ML libraries.
- We show the detailed taxonomies regarding the types,

TABLE I
STATISTICS OF EXPERIMENT SUBJECTS IN THIS STUDY

ML libraries	#CVEs	#Commits	# Vulnerability	Language
Tensorflow	36	1,197	250	C++/Python
PyTorch	N.A	563	75	C++/Python
Sickit-Learn	N.A	325	37	Python
Pandas	N.A	869	84	Python
Mlpack	N.A	664	46	C++
Numpy	N.A	1,198	149	C/Python
Scipy	N.A	793	42	C/Python
Overall	36	5,609	683	-

root causes, symptoms, and fixing patterns of security vulnerabilities in the studied ML libraries.

- We provide a set of practical guidelines to help machine learning development teams develop reliable and secure ML libraries.
- We release the dataset and source code of our experiments to help other researchers replicate and extend our study.

The rest of this paper is organized as follows.

Section II presents the methodology of this study. Section III presents the results. Section IV discusses the implication. Section V shows the threats to the validity of this work. Section VI presents the related studies. Section VII concludes this paper.

II. METHODOLOGY

A. Data Collection

1) Library selection: We design the following inclusion criteria for the selection of the studied ML libraries from GitHub: 1. Accessibility, i.e., the libraries should be popular, widely used, and open-source, 2. Maturity, i.e., the libraries should have been actively developed for a considerable amount of time, 3. Comprehensiveness, i.e., the libraries cover the current industrial machine learning practice and represent the critical aspects of machine learning developments, 4. Applicability, i.e., the libraries should support classical and state-of-the-art ML and DL models, data processing and manipulation, statistical model, etc., and 5. Availability, i.e., the libraries should have sufficient vulnerability fixing commits stored in their GitHub repositories. Initially, we selected the following libraries, i.e., Mlpack, TensorFlow, PyTorch, Numpy, Pandas, Scikit-learn, Pandas, Theano, Keras, Matplotlib, and Caffe. While Theano, Keras, and Caffe do not have sufficient vulnerability-fixing commits in their GitHub repositories. Matplotlib does not meet the applicability criteria. Eventually, using the criteria, we select seven ML libraries including TensorFlow [21], PyTorch [22], Scikit-learn [23], Mlpack [24], Pandas [25], Numpy [26], and Scipy [27] as our experiment subjects.

2) Vulnerability fixing commit collection: Our automatic fixing commit collection procedure is based on the rule heuristics implemented based on a set of security-related keywords extracted from [28]. The outcome of automatic filtering is 5,609 commits that are identified as vulnerability candidate commits. Note that such an automatic approach introduces false positives since the employed regex patterns are either too broad or not specific enough, leading to matches with commits that are not actually security-related. Hence we further started a manual labeling process to identify vulnerability-related commits.

3) Vulnerability-fixing commit identification: This subsection explains the adopted steps to filter out noises and identify vulnerability-related commits from the 5,609 commits extracted in the initial phase. We adopt the following steps:

Check possible CVE numbers: If there is any CVE number associated with a commit, we consider the commit as a vulnerability-related commit as CVE records are verified and confirmed by security professionals and researchers to track, analyze, and report on software vulnerabilities.

Analyze commit message and title: In this step, we look for specific security-related keywords in the title and the message of commits, e.g., *buffer overflow*, *stack overflow*, *RCE*, *exposure*, and *memory leaks*, etc.

Analyze code changes: We review the code changes associated with each commit to see if they address security-related vulnerabilities. For example, we check changes that involve adding or removing null checkers to determine whether the corresponding commit is related to fixing a null pointer dereference.

Analyze linked issues and pull requests: When a security vulnerability is discovered, it is typically reported through an issue on the project's issue tracker, which may include details about the vulnerability, such as how it can be exploited and what the potential impact is. Consequently, we review these linked issues to determine why a particular code change was made and whether it was made in response to a known security issue.

B. Data Labeling

In our study, we analyze each vulnerability-fixing commit from multiple aspects: 1) vulnerability type, 2) root cause, 3) symptom, and 4) fixing pattern. Please note that some existing studies on analyzing general software bugs in machine learning libraries have also provided taxonomies for these aspects [14], [16]. In this work, we did not adopt corresponding taxonomies from these studies. The reason is that existing studies mainly focus on the characteristics of general software bugs in ML libraries. As a result, it is not valid to adopt their classifications since general software bugs' characteristics and security vulnerabilities can be significantly different. In order to build vulnerability taxonomies for the studied ML libraries, we incrementally created them as we analyze and review each vulnerability-fixing commit [29]–[31]. For each commit, the first two authors analyze the commit data based on the defined research questions and take into account the following steps:

Analyze the vulnerability-fixing commits: We analyze the vulnerability-fixing commits data (commit title, message, code change, discussions, issues, and pull requests) to identify if there exist patterns and characteristics that are common among the vulnerabilities (details are in Section II-A3). We consider all 683 commits in the first round of reviewing.

Create preliminary taxonomies: Based on the analysis in the previous step, we create a preliminary taxonomy that categorizes the vulnerability-fixing commits based on their types, root causes, symptoms, and fixing patterns.

Expand taxonomies: We continue the previous step and assign new commits to the preliminary taxonomy. If we find a new commit that cannot fit into the preliminary taxonomy regarding its type, root cause, symptom, or fixing patterns, we expand the preliminary taxonomy by adding a new type created from the commit.

Note that, the authors worked together to reach a decision for all the disagreements during the above processes.

III. Result Analysis

In this section, we present and discuss our analysis results to address the four research questions we asked in Section I.

A. *RQ1: Vulnerability Types*

We organized vulnerability types into five high-level categories (shown in Figure 1) (i.e., *Memory*, *Numeric*, *Buffer*, *Resource*, *Concurrency*), involves more than 16 different Common Weakness Enumeration (CWEs)[1] covered by 621 (90.9%) of the 683 vulnerabilities. The remaining 62 (9%) vulnerabilities that appear infrequently and do not belong to any particular groups are included in the *Others* category.

Memory. Memory vulnerabilities happen when ML library developers improperly handle memory allocation and deallocation. This category accounts for 226 (33%) of the vulnerabilities. Specifically, it contains the following five types of CWEs: 1) *Missing Release of Memory after Effective Lifetime (Memory Leak (CWE-401)*, 2) *Null Pointer Dereference (CWE-476)*, 3) *Infinite Loop (CWE-835)*, 4) *Double Free (CWE-415)*, and 5) *Use After Free (CWE-416)*.

Numeric. ML libraries perform a large number of tensor-level computations, more specifically dealing with floating point operations, which results in *Numeric* vulnerabilities. This category accounts for 198 (28.9%) of the vulnerabilities. It mainly has four types of CWEs: 1) *Integer Overflow (CWE-190)*, 2) *Insufficient Precision or Accuracy of a Real Number (CWE-1339)*, 3) *Division by Zero (CWE-369)*, and 4) *Integer Underflow (CWE-191)*.

Buffer. Buffer vulnerabilities also known as buffer overflows or overruns, occur when a computer writes more data to a temporary storage space in memory known as a buffer than it can manage. This category accounts for 96 (14%) of the vulnerabilities. It mainly covers five types of CWEs: 1) *Out of Bound Read (CWE-125)*, 2) *Stack Overflow (CWE-121)*, 3) *Heap Buffer Overflow (CWE-122)*, 4) *Buffer Overflow (CWE-120)*, and 5) *Out of Bound Write (CWE-787)*.

[1]CWE (https://cwe.mitre.org/) is a well-known and widely accepted standard for documenting and categorizing security vulnerabilities. The use of CWE categories can assist in standardizing vulnerability descriptions and make it easier for researchers and practitioners to comprehend and explain the nature of security issues.

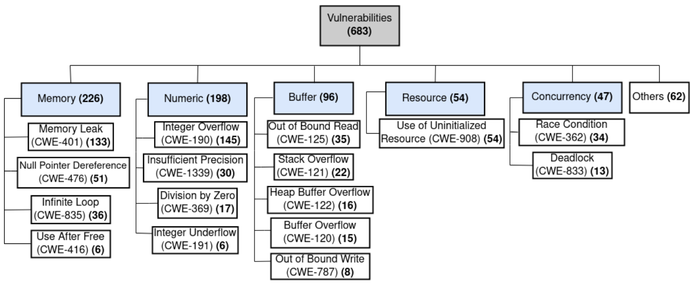

Vulnerabilities (683)
Memory (226)
Memory Leak (CWE-401) (133)
Null Pointer Dereference (CWE-476) (51)
Infinite Loop (CWE-835) (36)
Use After Free (CWE-416) (6)
Numeric (198)
Integer Overflow (CWE-190) (145)
Insufficient Precision (CWE-1339) (30)
Division by Zero (CWE-369) (17)
Integer Underflow (CWE-191) (6)
Buffer (96)
Out of Bound Read (CWE-125) (35)
Stack Overflow (CWE-121) (22)
Heap Buffer Overflow (CWE-122) (16)
Buffer Overflow (CWE-120) (15)
Out of Bound Write (CWE-787) (8)
Resource (54)
Use of Uninitialized Resource (CWE-908) (54)
Concurrency (47)
Race Condition (CWE-362) (34)
Deadlock (CWE-833) (13)
Others (62)

Fig. 1. The taxonomy of vulnerability types studied in this work.

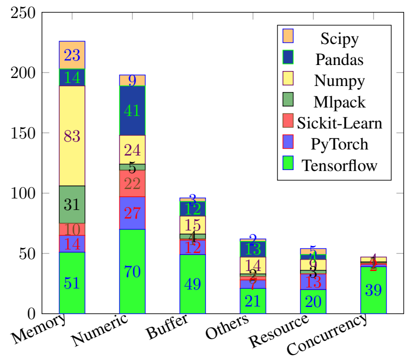


Fig. 2. The distribution of software vulnerabilities in the studied ML libraries.

Resource. Resource vulnerabilities occur when a program fails to manage resources such as memory, and file handles. This category accounts for 54 (7.9%) of the vulnerabilities. It consists of only one subcategory: *Use of Uninitialized Resource (CWE-908)*

Concurrency. Concurrency in ML libraries is a type of software weakness that occur due to their complexity, a large number of computations that are not thread-safe, and asynchronous execution of computation graphs. This category accounts for 47 (6.8%) vulnerabilities. It consists of two types of CWEs: 1) *Race Condition (CWE-362)* and 2) *Deadlock (CWE-833)*.

*1) **Implication**:* ML libraries use large tensors to represent data, which can consume a lot of memory. These intensive usages of tensors require frequent allocation and deallocation of memory in the implementation of ML libraries (memory management) leading to vulnerabilities such as buffer overflow and memory leaks. Buffer overflows occur when data is written to a buffer more than it can contain, and can cause data to overflow into neighboring memory areas, potentially overwriting important data or executing malicious code. As shown in Table II, *Memory Leak* and *Null Pointer Dereference* are the two most common subcategories of *Memory* vulnerabilities. *Memory Leak* is common in the Numpy library where memory management should be done manually due to the development of language barriers. To make the matter more concrete, we elaborate on an example of *Memory Leak* [2] where the

[2]https://github.com/numpy/numpy/commit/4e19f408de900f958441af4ec8a458f5ce6473eb

developer has created a slice object but never released by *decref* statement. According to the Python C API reference documentations[3], if a created object is going to be returned by the function in which it is created, the reference to the object should be decremented.

ML libraries frequently perform calculations on the size of arrays, multiplication of tensors together, or multiplying or adding integer values. For example, ML libraries may use a 32-bit integer type to store the result of a multiplication operation that produces a value greater than $2^31 - 1$ (the maximum value that can be represented by a signed 32-bit integer), an integer overflow will occur and the value will wrap around to a negative value. This can result in numerical instability such as leading to poor model performance or crash. Attackers can exploit the ranges of the defined variables and cause a denial of service attacks or make the result of model training and inference inconsistent. An example of integer overflow vulnerability found in the Tensorflow library[4] where the developer has defined integer variables with 32 bits precision which cause overflow. The suggested fix is defining integer variables with higher precision, e.g., in this commit, the author has defined 64 bits integer variables.

2) ***Comparison with traditional software****:* We further check whether there is a similar trend regarding the distribution of security vulnerability types in traditional software systems or not. We based our comparison according to the well-known paper published on characteristics of traditional software systems [32]. According to Tan et al. [32], *Memory Leak (CWE-401)* is the most frequent vulnerability type which is similar to our finding and confirms that *Memory Leak (CWE-401)* is the dominant vulnerability type in both the studied ML libraries and general software systems. We also find that *Integer Overflow (CWE-190)* is the dominant vulnerability type in the category of *Numeric*. This trend is not similar to general software systems [32] and further confirms our findings that *Numeric* is significant to the studied ML libraries as they rely on heavy computations using tensors and arrays, based on floating point operations. We further check a recently publish paper [33] which characterizes bugs in web assembly compilers. They find that *Data type incompatibility* accounts for 15.75% of bugs in web assembly compilers. These bugs arise from incompatibility in interfaces between WebAssembly and JavaScript. This pattern is not similar to *Numeric* vulnerabilities in the studied ML libraries as they have very sophisticated vulnerability patterns.

Finding 1: Memory and **Numeric** are the dominant categories of vulnerability types across the studied ML libraries, accounting for 33% and 28.9% of vulnerabilities.

[3]https://docs.python.org/3/c-api/memory.html

[4]https://github.com/tensorflow/tensorflow/commit/087859fce9409991164f727735743da4cb310fd4

TABLE II
SUBCATEGORIES OF **MEMORY**. **ML** DENOTES MEMORY LEAK, **NPD** IS NULL POINTER DEREFERENCE, **IL** IS INFINITE LOOP, AND **UAF** IS USE AFTER FREE.

Library	ML	NPD	IL	UAF	Overall
Tensorflow	6	25	19	1	51
PyTorch	2	7	5	0	14
Sickit-Learn	8	0	2	0	10
Mlpack	26	3	2	0	31
Pandas	5	6	3	0	14
Numpy	64	10	5	4	83
Scipy	22	0	0	1	23

TABLE III
SUBCATEGORIES OF **NUMERIC**. **IO** IS INTEGER OVERFLOW, **IP** IS INSUFFICIENT PRECISION, **DZ** IS DIVISION BY ZERO, AND **IU** IS INTEGER UNDERFLOW.

Library	IO	IP	DZ	IU	Overall
Tensorflow	63	4	2	1	70
Pytorch	18	6	3	0	27
Sickit-Learn	8	4	7	3	22
Mlpack	2	2	1	0	5
Pandas	34	3	3	1	41
Numpy	13	10	1	0	24
Scipy	7	1	0	1	9

B. RQ2: Root Causes

Root causes in the studied ML libraries are organized into five high-level categories (illustrated in Figure 4) including *Data Type*, *Memory*, *API*, *Business Logic*, and *Concurrency* covered by 619 (90.6%) of the 683 vulnerabilities. The remaining 64 (9.3%) root causes have no clear indication about their types, and hence we group them in *Others* category. Note that although some of the high-level taxonomy names from vulnerability types (RQ1) and root causes (RQ2) are similar, they refer to different levels of abstraction in Figure 1 (RQ1) and Figure 4 (RQ2). For example, In RQ1, the *Memory* category refers to a broader range of memory vulnerabilities. In RQ2, *Memory* is a more specific category that explains the root cause of vulnerabilities depicted in Figure 1.

Data Type. This root cause category accounts for 215 (31.4%) of vulnerabilities. Subcategories are including 1) *Lack of Validating Tensors and Arrays Property*: Sometimes implementations fail to check tensors or array properties, e.g. shape or rank. As a result, attackers can exploit tensor or arrays properties and craft special inputs to trigger denial of service via segmentation fault or overflows, 2) *Using Improper Data Type*: When developers have confusion about what kind of data types should they use for variables, e.g. using a *float* instead of a *double*, 3) *Integer Variable Range Issue*: When developers define integer variables with limited range or very large range, e.g. defining *int32* bits instead of *int64* bits, 4) *Lack of Overflow Checking*: If the program does not check for overflow, the result of the operation may be incorrect, which can lead to vulnerabilities in ML libraries, 5) *Float Variable Range Issue*: Similar to *Integer Variable Range Issue*, for example using float32 instead of using float64 data type for numeric operations, 6) *Out of Bound Nanoseconds for Timestamp*: Generally, pandas timestamp data type represents date object in nanosecond resolution which failure in storing

date in its default range results in out of bound vulnerability.
Memory. This root cause category accounts for 30.4% of vulnerabilities. The subcategories are including 1) *Improper Memory Management:* When a developer has confusion in memory management, either misuses a memory release statement or forgets to release memory after its effective lifetime, 2) *Invalid Memory Access:* When a process tries to access memory locations filled with null values, corrupted, already been deleted, or freed, 3) *Very Deep Tensors and Computation Graphs*: Sometimes tensors or computation graphs grow unexpectedly at runtime and exceed stack or buffer size which results in a segmentation fault, 4) *Uninitialized Resource*: An uninitialized resource vulnerability occurs when a program or application fails to properly initialize a resource, such as a variable or a file, before using it. This can lead to unexpected behavior and potentially allow an attacker to exploit the vulnerability and execute malicious code or steal sensitive information.
Business Logic. This category of the root causes accounts for 95 (13.9%) of vulnerabilities. It includes 1) *Improper Exception Handling*: When developers incorrectly handle exceptional conditions leading to termination of the software during normal executions of the software, 2) *Incorrect Index Calculation*: Normally, tensors or arrays are accessed by indices, however, sometimes indices are vulnerable to many attacks, e.g. assigning a large value to an index, which cause array out of bound access vulnerability and trigger a denial of service via crash or segmentation fault, 3) *Wrong Order of Execution*: When the execution order of instructions in the backend implementation is not in the intended order. This can lead to unexpected behavior, data corruption, or crashes.
API. This category accounts for 71 (10.3%) of total records. Subcategories include 1) *API Misuse*: When developers mistakenly use a specific API, e.g., passing parameters in wrong orders, lack of using optional parameters, and mistakenly using optional parameters, 2) *Using Wrong API*: When developers mistakenly use improper APIs, e.g. using *numpy.empty()* instead of using *numpy.empty_like()*, 3) *Malicious Parameters*: When developers pass malicious or invalid parameters to API calls that are exploitable by attackers. Attackers can exploit these parameters by crafting particular inputs to take control of ML libraries, and 4) *Third Party Library Issue*: When developers mistakenly use either wrong versions of APIs or outdated ones.
Concurrency. This category involves concurrent access of resources in a shared environment by multiple threads due to improper resource locking, releasing, or simultaneous resource access accounting for 30 (4.3%) vulnerabilities. Subcategories are 1) *Missing Locking Statement*: When developers forget to lock resources which mostly results in race condition or deadlock vulnerabilities, 2) *Improper Usage of Locking Statement*: When developers use locking statements improperly on resources, or they release locked resources inappropriately, which may result in deadlock or race condition vulnerabilities, and 3) *Improper Resource Locking*.

Figure 3 shows the distribution of root causes of vulnerabilities in different libraries. As can be seen, the *Data Type* category is the most common root cause of vulnerabilities across the studied ML libraries. *Memory* is the second most common root cause of vulnerabilities across the studied ML libraries. Table IV and Table V further shows the distribution of subcategories of *Data Type* and *Memory* across the studied ML libraries respectively. As we can see, *Lack of Validating Tensors or Arrays Property* and *Integer Variable Range Issue* are the major subcategories in *Data Type*. While *Improper Memory Management* and *Invalid Memory Access* are the dominating subcategories in *Memory*.

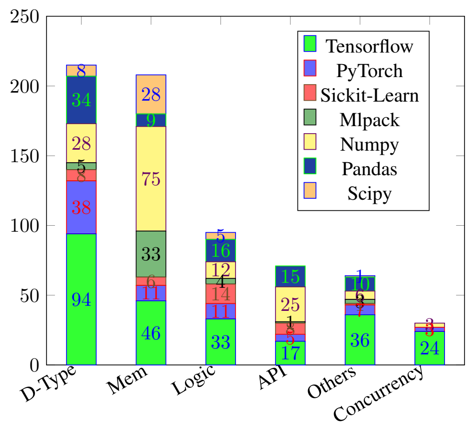
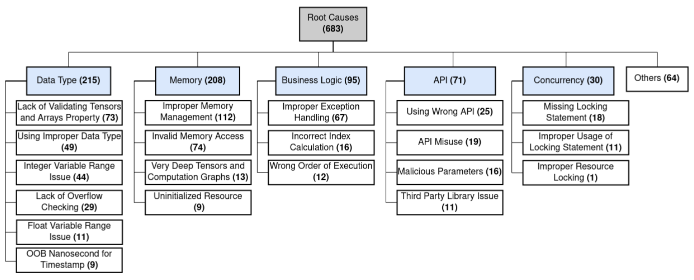


Fig. 3. Distribution of root causes across libraries



Root Causes (683)
Data Type (215)
Lack of Validating Tensors and Arrays Property (73)
Using Improper Data Type (49)
Integer Variable Range Issue (44)
Lack of Overflow Checking (29)
Float Variable Range Issue (11)
OOB Nanosecond for Timestamp (9)
Memory (208)
Improper Memory Management (112)
Invalid Memory Access (74)
Very Deep Tensors and Computation Graphs (13)
Uninitialized Resource (9)
Business Logic (95)
Improper Exception Handling (67)
Incorrect Index Calculation (16)
Wrong Order of Execution (12)
API (71)
Using Wrong API (25)
API Misuse (19)
Malicious Parameters (16)
Third Party Library Issue (11)
Concurrency (30)
Missing Locking Statement (18)
Improper Usage of Locking Statement (11)
Improper Resource Locking (1)
Others (64)

Fig. 4. Taxonomy of root causes in the studied ML libraries.

*1) **Implication**:* Tensors and arrays have properties such as shape, rank, axis or dimensions, and size which can be vulnerable to attacks from external attackers. These vulnerabilities can be exploited by passing invalid values, resulting in out-of-bounds read and denial of service through a segmentation fault or crash. An intuitive example of such attacks is this vulnerability from Tensorflow library[5] where *tf.raw_ops.QuantizeAndDequantizeV2* allows the axis argument to accept invalid arguments. In this example, the constraint is that *axis_* accepts -1 and the checker (*OP_REQUIRES*) is true as long as the value is less than or equal to -1, though it results in heap underflow vulnerability which is exploitable by attackers to write or read their data on top of the heap.

[5]https://github.com/tensorflow/tensorflow/commit/c5b0d5f8ac19888e46ca14b0e27562e7fbbee9a9

TABLE IV
SUBCATEGORIES OF **DATA TYPE ERRORS**. **LVTAP** IS LACK OF VALIDATING TENSORS AND ARRAYS PROPERTY, **IVRI** IS INTEGER VARIABLE RANGE ISSUE, **UIDT** IS USING IMPROPER DATA TYPE, **LOC** IS LACK OF OVERFLOW CHECKING, **FVRI** IS FLOAT VARIABLE RANGE ISSUE, AND **ONT** IS OUT OF BOUND NANOSECOND FOR TIMESTAMP.

Library	LVTAP	IVRI	UIDT	LOC	FVRI	ONT	Overall
Tensorflow	42	21	20	7	4	0	94
Pytorch	14	8	8	4	4	0	38
Sickit-Learn	0	3	1	3	1	0	8
Mlpack	2	0	2	0	1	0	5
Pandas	1	8	8	7	1	9	34
Numpy	14	4	5	5	0	0	28
Scipy	0	0	5	3	0	0	8

TABLE V
SUBCATEGORIES OF **MEMORY ERRORS**. **IMM** IS IMPROPER MEMORY MANAGEMENT, **IMA** IS INVALID MEMORY ACCESS, AND **VDTC** IS VERY DEEP TENSORS AND COMPUTATION GRAPHS, AND **UR** IS UNINITIALIZED RESOURCE.

Library	IMM	IMA	VDTC	UR	Overall
Tensorflow	3	35	8	0	46
Pytorch	1	9	1	0	11
Sickit-Learn	6	0	0	0	6
Mlpack	21	5	0	7	33
Pandas	3	5	1	0	9
Numpy	57	13	3	2	75
Scipy	21	7	0	0	28

ML library developers use statistically typed languages for development which can be confusing for developers, as they must decide the range of integer or float variables. A very high precision range can result in performance degradation and resource consumption, while a low precision range can cause loss and unexpected behavior in the ML libraries. These difficulties can lead to mistakes by developers and may lead to defining variables with improper range precision. For example, the default integer variable in the Tensorflow library has 32 bits, we observed that defining *int32* bits is one of the major root causes of vulnerabilities. For example, in this commit from Tensorflow library[6], the developer has defined *output_elements* as a *const int32* bits. The cause *CalculateTensorElementCount()* to assign a very large value in a 32 bits integer variable and cause integer overflow. The suggested fix is using *int64* range precision to prevent the vulnerability.

Finding 2: Data Type category is the most common root cause accounting for 31.4% vulnerabilities in the studied ML libraries. This is because developers have confusion about the range of integer or float variables during the development process. High precision ranges can lead to performance degradation and resource consumption, while low precision ranges can cause loss and unexpected behavior in the model.

*2) **Comparison with traditional software**:* We start by investigating the difference between ML libraries versus traditional software in terms of *Data Type* and *API* as two important root causes of vulnerabilities in the studied ML libraries.

Similar to the findings in *RQ1*, we find that *Data Type* category is the dominant category of root causes in the studied ML libraries, with *Lack of Validating Tensors and Arrays Property* as the most frequent subcategory. Due to this dominance, we further check whether root causes in traditional software systems follow the same pattern or not. Consequently, we compare our findings with the findings in [32] and [34]. Hirsch and Hofer [34] analyze the root cause of vulnerability reports gathered from 103 Github projects. According to our investigations, we find that in both papers, semantic vulnerabilities are the major dominant root cause of vulnerabilities accounting for 87% (maximum percentage which belongs to Mozilla project[7]) and 269/512 (52.5%) in [34] and [32]. As a result, the root cause patterns in traditional software are not similar to patterns of root causes in the studied ML libraries.

API vulnerabilities in the studied ML libraries and general software are distinct in terms of their nature and impact. One distinction is that ML libraries are frequently used for more sophisticated tasks like image or speech recognition, which necessitate a greater level of precision and accuracy. Additionally, ML libraries often use third-party dependencies (such as numerical libraries such as FP16, absl, fft2d in TensorFlow or BLAST, and LAPACK in Numpy) in order to perform numerical computations. If these dependencies are not properly maintained or updated, it can cause vulnerabilities in the ML library. For example, if a security vulnerability is discovered in a third-party library used by an ML library, it can cause the ML library to be vulnerable to attacks as well. Overall, API vulnerabilities in ML libraries are more complicated and difficult to solve than in traditional software, although the development process and debugging methodologies are similar.

We further investigate the difference between API vulnerabilities in the studied ML libraries versus traditional software highlighted by other researchers. According to a current study on *API* vulnerabilities conducted by Amann et al., [35], missing and redundant API calls are the most frequent vulnerabilities. At the same time, these are the least vulnerabilities in the studied ML libraries. We conclude that developers of the studied ML libraries have difficulty understanding which APIs they should use, how to use them, and make them secure.

Finding 3: Lack of Validating Tensors and Arrays Property is the most frequent root cause of vulnerabilities in the studied ML libraries. If the properties of tensors and arrays are not properly validated, it can lead to a crash, segmentation fault, or unexpected behavior in the model. This can happen because of a lack of proper validation checks in the library's backend implementation.

[6]https://github.com/tensorflow/tensorflow/commit /087859fce9409991164f727735743da4cb310fd4

[7]In [32], semantic vulnerabilities account for 82.5% and 70.1% in Apache and Linux projects, respectively.

TABLE VI
SUBCATEGORIES OF **API ERRORS**. **AM** IS API MISUSE, **MP** IS MALICIOUS PARAMETERS, **TPLI** IS THIRD PARTY LIBRARY ISSUE, AND **UWA** IS USING WRONG API.

Library	AM	MP	TPLI	UWA	Overall
Tensorflow	2	2	6	7	17
Pytorch	1	1	1	2	5
Sickit-Learn	3	2	0	3	8
Pandas	4	7	1	3	15
Numpy	9	4	2	10	25
Mlpack	0	0	1	0	1
Scipy	0	0	0	0	0

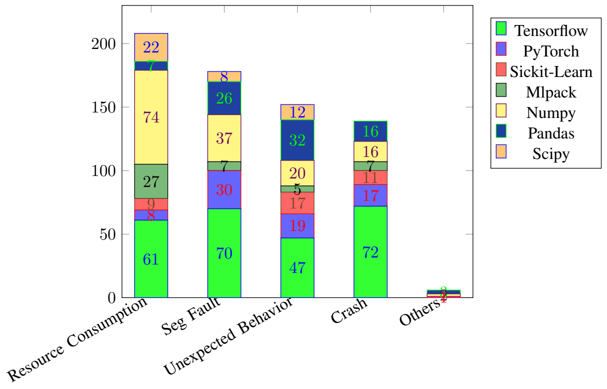


Fig. 5. Distribution of symptoms across different libraries.

Finding 4: API in the studied ML libraries have distinct vulnerability patterns compared to traditional software which makes testing and debugging them difficult and time-consuming.

C. RQ3: Symptoms

Symptoms of vulnerabilities in the studied ML libraries are organized into 4 categories including *Segmentation Fault*, *Crash*, and *Unexpected Behaviour*, *Resource Consumption*, covered by 677 (99.1%) vulnerabilities. The remaining 6 vulnerabilities have no clear indication about their outcome, and hence we group them in *Others* category.

Resource Consumption: Accounting for 30.45% (208) of vulnerabilities, is the exhaustion of available resources, e.g., increasing memory usage because of uncontrolled or improper allocation of the memory.

Segmentation Fault: Accounting for 26% (178) of vulnerabilities, when a program outputs *Segmentation fault* or *Segmentation fault (core dumped)* in the output which is the symptom for segmentation fault[8].

Unexpected Behavior: Accounting for 22.2% (152) of vulnerabilities, happens when the library produces results that are not expected. For example, in this *Integer Overflow* vulnerability from Scikit-learn library[9] where $pk * qk$ returns *inf* instead of *float* because of exceeding int32 bits limits during multiplication.

[8]https://stackoverflow.com/questions/49092527/illegal-instructioncore-dumped-tensorflow

[9]https://github.com/scikit-learn/scikit-learn/commit/622f912095308733ddfe572a619b1574b9da335e

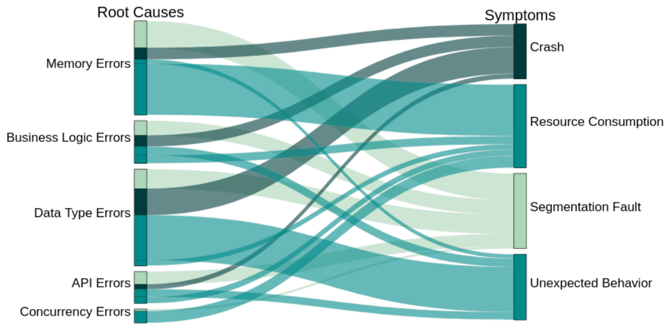


Fig. 6. Mapping of root causes to symptoms.

Crash: Accounting for 20.3% (139) of vulnerabilities, happens when a program or application terminates unexpectedly, is called *Crash* also known as *crashing*. This can happen for a variety of reasons, such as accessing an invalid memory location, a divide-by-zero vulnerability, or an infinite loop.

Figure 5 demonstrates the distribution of symptoms across different libraries. As you can see, the most frequent symptom is *Resource Consumption* accounting for 30.45% of vulnerabilities. We also draw a mapping from root causes to symptoms to interpret what is the outcome of vulnerabilities as shown in Figure 6. It is observable from Figure 6 that the majority of vulnerabilities caused by *Memory* have *Resource Consumption* as their impact. Also, the majority of vulnerabilities caused by *Data Type* have *Unexpected Behavior* as their impact. One possible usage scenario of vulnerability symptoms in the studied ML libraries is that ML developers can narrow down the types of vulnerabilities present in their codebase and take steps to fix them. Also, ML developers do not need to develop test oracles for unit testing of the studied ML libraries in order to understand vulnerabilities that have *Segmentation Fault* and *Crash* as their symptoms.

Finding 5: Resource Consumption and **Segmentation Fault** and are the most common symptoms of vulnerabilities accounting for 30.45% and 26% of vulnerabilities respectively. The studied ML libraries are vulnerable to resource consumption vulnerabilities because they are designed to perform computationally intensive operations on large amounts of data. These operations can consume a significant amount of memory and processing power, making it easy for vulnerabilities to cause the program to consume too many resources and crash or become unresponsive.

1) Comparison with traditional software: We find that *Resource consumption* and *Segmentation Fault* are two dominant symptoms of vulnerabilities in the studied ML libraries. There is still an important question that needs to be addressed: *What is the difference between symptoms in traditional software systems and ML libraries?* To answer this question, we compare our findings with the study conducted by Tan et al. [32]. According to the findings in [32], *Functional* impact is the major symptom of bugs in traditional projects which greatly deals with the fact that the software system is not functioning according to its specifications. However, regarding ML libraries, we have a completely different trend where the majority of symptoms of vulnerabilities in the studied ML libraries result in the crashing of the library during runtime via *Segmentation fault* which is the dominant symptom.

D. RQ4: Fixing Patterns

Fixing patterns are organized into six high-level categories (shown in Figure 7) including *Add Checkers*, *Resolve Data Type Vulnerabilities*, *Resolve Memory Vulnerabilities*, *Resolve API Vulnerabilities*, *Resolve Concurrency Vulnerabilities*, and *Modify Business Logic Errors* covered by 586 (85.7%) of vulnerabilities. The remaining 97 (14.2%) fixing patterns have no clear indication about their types and hence are included in the *Others* category.

Add Checkers. Fixing patterns in this category mainly relate to the addition of either library-specific or conventional checkers to fix vulnerabilities, accounting for 155 (22.6 %) vulnerabilities. Subcategories are 1) *Add Checker for Tensors and Arrays Property*: This is the most common fixing pattern where developers use if conditions or library-specific checkers to validate tensor and arrays properties, e.g., shapes, ranks, values, or elements, 2) *Add Checker for Null Pointer Dereference:* Developers often add checkers either using if conditions or library-specific checkers to prevent null pointer dereferences, 3) *Add Checker for Overflow*: This fixing pattern is mainly used to fix overflow vulnerabilities where developers either add if modules or library-specific checkers to prevent overflow or throw appropriate exceptions.

Resolve Memory Vulnerabilities. Fixing patterns in this category relates to memory management efforts, which fix 130 (19%) of total vulnerabilities. Subcategories are 1) *Manage Memory Release*: is used to fix vulnerabilities related to incorrect or inappropriate memory allocations and 2) *Resource Initialization*: when developers initialize tensors or variables to fix vulnerabilities.

Modify Business Logic: It is accounting for 16.8% of vulnerabilities (115), related to improving exception handling, file handling, control flows, methods, or classes having incorrect logic. Subcategories are 1) *Improved Exception Handling*: When a program crashes, it is necessary to raise appropriate messing to help developers with debugging operations. This pattern adds missing error reporting or modifying existing exception handling, 2) *Modify Index Calculation*: This pattern is used to fix vulnerabilities caused by improper calculation of tensors and arrays indices, 3) *Avoid Overflow on Deep Tensors and Graphs*: This pattern is used when developers try to prevent overflow caused by small stack size or deep computation graphs created in runtime, 4) *Modify Order of Execution*: Developers change the location of semantically related statements to fix vulnerabilities, 5) *Close File Handler*: It is used to fix file descriptor leak vulnerability where developers sometimes forget to close opened files after its lifetime.

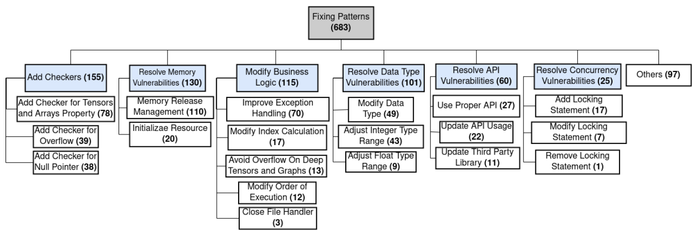

Fixing Patterns (683)
Add Checkers (155)
Add Checker for Tensors and Arrays Property (78)
Add Checker for Overflow (39)
Add Checker for Null Pointer (38)
Resolve Memory Vulnerabilities (130)
Memory Release Management (110)
Initializae Resource (20)
Modify Business Logic (115)
Improve Exception Handling (70)
Modify Index Calculation (17)
Avoid Overflow On Deep Tensors and Graphs (13)
Modify Order of Execution (12)
Close File Handler (3)
Resolve Data Type Vulnerabilities (101)
Modify Data Type (49)
Adjust Integer Type Range (43)
Adjust Float Type Range (9)
Resolve API Vulnerabilities (60)
Use Proper API (27)
Update API Usage (22)
Update Third Party Library (11)
Resolve Concurrency Vulnerabilities (25)
Add Locking Statement (17)
Modify Locking Statement (7)
Remove Locking Statement (1)
Others (97)

Fig. 7. Taxonomy of fixing patterns in ML libraries.

Resolve Data Type Vulnerabilities: Fixing patterns in this category focus on resolving vulnerabilities related to data types, which cover 14.7% (101) of vulnerabilities. Subcategories include 1) *Adjust Integer Type Range*: This pattern is used to either increase or decrease the range precision of integer variables to prevent numerical overflow vulnerabilities, 2) *Modify Data Type*: When data types are defined incorrectly, this pattern alter existing defined data types to fix corresponding vulnerabilities, 3) *Adjust Float Type Range*: Similar to *Adjust Integer Type Range*, this pattern adjusts float variables range precision, e.g., either increase the range precision or decrease it.

Resolve API Vulnerabilities: Fixing patterns in this category are mainly used to fix vulnerabilities introduced by inappropriate API usages, which help fix 60 (8.7%) of vulnerabilities studied in this paper. The detailed subcategories are 1) *Using Proper API*, 2) *Update API Usage*, and 3) *Update Third Party Library*.

Resolve Concurrency Vulnerabilities. Accounting for 25 (3.6%) vulnerabilities, this category of fixing pattern is used to fix vulnerabilities related to concurrency issues resulting in deadlock or race condition errors. Subcategories include 1) *Add Locking Statement*, 2) *Modify Locking Statement*, and 3) *Remove Locking Mechanism*.

*1) **Implication**:* We find that developers of the studied ML libraries frequently use validation checks in the code, such as checking the shape and data type of tensors and arrays before using them in computations. In this example[10] which is *Integer Overflow*, there is no checker on *data[axis]* to make sure it does not exceed int32 bits range limits. The developer overcomes the problem by adding a checker of *TF_LITE_ENSURE* on line 76 in *tensorflow/lite/kernels/concatenation.cc*. We also find that developers of the studied ML libraries usually mitigate *Data Type vulnerabilities* by explicitly setting data types when creating tensors, for example, using *tf.float32* instead of *tf.float* for a tensor.

[10]https://github.com/tensorflow/tensorflow/commit/4253f96a58486ffe84b61c0415bb234a4632ee73

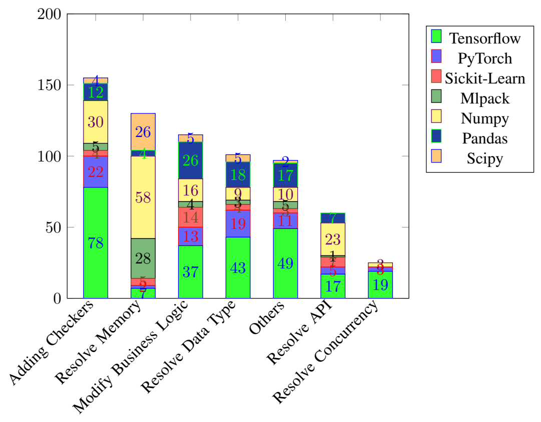


Fig. 8. Distribution of fixing patterns across different libraries.

TABLE VII
Subcategories of **Add Checkers**. **ACTAP** is Add Checker for Tensors and Arrays Property, **ACO** is Add Checker for Overflow, and **ACNP** is Add Checker for Null Pointer Dereference.

Library	ACTAP	ACO	ACNP	Overall
Tensorflow	45	14	19	78
Pytorch	14	5	3	22
Sickit-Learn	0	3	1	4
Mlpack	2	0	3	5
Pandas	2	9	1	12
Numpy	14	5	11	30
Scipy	1	3	0	4

Finding 6: **Adding Checkers for Tensors and Array Properties** is the most common fixing pattern used by developers accounting for 11.4% of vulnerabilities. In this pattern, developers often implement solutions such as adding validation checks either using library-specific macro checkers or general if statements to guard against weaknesses corresponding to lack of validation.

IV. Discussion

Our study reveals several interesting findings that can serve as practical guidelines for both industry and academic communities to improve software security development for the studied ML libraries.

A. *Implications to ML Community*

Adjust Data Types Range Precision. It is important to select the appropriate data type for representing numerical data in the backend implementation of the studied ML libraries, based on the range and precision requirements of the computation being performed, otherwise, it results in: *Numerical instability*: When the range of the data types is too large or too small, the model can become numerically unstable, leading to inaccuracies and errors in the computations, *Overflow and underflow*: When the range of the variables exceeds the maximum or minimum representable value, overflow or underflow can occur, resulting in incorrect and unexpected results, *Loss of precision*: When the range of data types is not set to the appropriate precision, the model may lose precision in the computations, leading to inaccuracies in the results. To mitigate these issues, developers can adjust the range precision of the data types explicitly during the development of the backend of the studied ML libraries.

Validate Properties of Tensors and Arrays. There are various vulnerabilities in the studied ML libraries related to the characteristics of tensors and arrays. *Data type mistakes*: When tensor and array attributes, such as shape and data type, are not correctly validated, it can lead to data type problems, such as attempting to execute operations on tensors with incompatible data types. *Memory Errors*: When tensor and array characteristics such as size and shape are not handled correctly, memory issues such as buffer overflow or out-of-memory errors might occur. *Indexing issues*: When tensor and array attributes, such as shape and dimension, are not validated properly, it can result in indexing issues, such as attempting to access an out-of-bounds element.

Improve Static Checkers. Even though the existing static checkers [36]–[41] have been widely adopted to detect vulnerabilities on traditional projects [42]–[44], it is unclear how well they detect real-world security vulnerabilities in the studied ML libraries. Current static checkers are mostly equipped with simple rules to detect security vulnerabilities in general software projects, while our root causes analysis (details are in Section III-B) shows that the studied ML security vulnerabilities are complex in nature. For example, we find that one of the major root causes of security vulnerabilities in the studied ML libraries is the lack of validating tensors and array attributes which results in severe vulnerabilities such as buffer overflow or integer overflow. As a result, it is critical to extending existing checkers to support security vulnerabilities in the studied ML libraries. One promising direction is to extend the built-in rule database of static checkers by including ML-specific APIs and macro checkers, which can be an effective way to improve the accuracy of vulnerability detection in ML libraries.

Use Fuzz Testing to Detect ML Security Vulnerabilities. We find that *Lack of Validating Tensors and Arrays Property* is one of the major root causes of security vulnerabilities in the studied ML libraries, because of which, external attackers or end-users of ML APIs intentionally or unintentionally send malformed input to the backend implementation of the studied ML libraries to trigger these vulnerabilities. Fuzz testing [45], i.e., an automated testing approach that injects malformed inputs into a system to reveal software vulnerabilities, can be an effective approach to find edge cases that cause security vulnerabilities in the implementation of the studied ML libraries.

Mitigate Memory Errors. There are several ways to prevent memory-related vulnerabilities in the studied ML libraries. *Memory Management*: Due to development language barriers, the memory management in the backend implementation of the studied ML libraries should be done manually. Manual memory management is critical since it may result in memory leak vulnerability leading to a gradual depletion of system resources and potential crashes or security vulnerabilities. It is important for developers to follow secure coding practices when implementing memory management in the studied ML libraries. This includes using secure memory allocation and deallocation functions, checking for buffer overflows and other memory-related vulnerabilities, and regularly testing the code for potential memory leaks. *Exception handling*: Use exception handling to catch and handle any errors that occur due to memory issues, such as out-of-memory errors, *Use Profiling Tools*: Use memory profiling tools to identify and diagnose memory issues, such as memory leaks and excessive memory usage.

B. Significance of Our Findings

Our work makes significant contributions to the studied ML community since it addresses fundamental challenges regarding the security and reliability of the studied ML libraries. This study gives insights into the prevalent challenges that the studied ML libraries encounter by investigating the types of vulnerabilities, root causes, symptoms, and repair patterns. In particular, characterizing vulnerability types help developers in identifying and prioritizing possible security issues, demystifying the core causes of these vulnerabilities can aid in the construction of more secure and trustworthy ML libraries, and understanding vulnerability symptoms and repairing patterns can assist developers in promptly identifying and addressing security vulnerabilities in the studied ML libraries.

V. Threats to validity

Construct validity Our study is primarily aimed at identifying and examining vulnerabilities present in ML libraries, rather than evaluating their severity. Although CVSS severity data can provide important information about vulnerabilities, it is important to note that other factors can also contribute to their impact. Therefore, our focus is on identifying and classifying vulnerabilities based on their types, root causes, symptoms, and fixing approaches as they relate to the functionality of the ML libraries, rather than assigning severity scores.

Internal Validity. The main internal threat to our work is our manual analysis labeling and classification of software security vulnerabilities which may suffer from subjective bias and errors. To guard against this, two first authors who are skilled in software security vulnerabilities have reviewed the collected commits. Authors also discuss any possible disagreement until a consensus is reached.

External Validity. The dominant threat to the external validity of this study is the collected dataset. To overcome this threat, we collected commits from seven different ML libraries; two are very famous and widely used DL libraries, including TensorFlow and PyTorch; two of them are Mlpack and scikit-learn which is renowned classical ML library which often is used beside DL libraries. We also collected data from three well-known data analytical and visualization tools, including Pandas, Numpy, and Scipy. The reason behind this diverse data collection is to generalize our findings to wide domains and increase the reliability of findings.

VI. Related Work

A. Studies on General Bugs

There are many efforts to characterize software security vulnerabilities in traditional software systems [32], [46]–[51]. Tan et al. [32] conducted an empirical study on three notable projects, including Linux kernel, Mozilla, and Apache, via analyzing around 2k real-world bugs. They revealed that semantic bugs are the major common bugs in general software systems, and memory bugs decrease as they evolve. The significant difference with our analysis is that they did not introduce vulnerability types; instead, they focused on root causes analysis. Also, their analysis is based on general and vulnerable related bugs, while we do not cover the general bugs. Bosu et al. [46] analyzed code review requests from 10 software projects to identify vulnerable code changes. They find that *Race Condition* and *Buffer Overflow* are the most common vulnerability types in traditional software systems. These findings are not aligned with our work where *Race Condition* and *Buffer Overflow* are the least common vulnerability types.

B. Studies on ML Bugs

1) Studies on ML API Usage Bugs: Islam et al. [14] conducted the first empirical study on API usage bugs of five DL libraries, including Caffe, Keras, TensorFlow, Theano, and Torch. They collected data from StackOverflow posts, and Github commits to perform their manual analysis. The authors analyzed bug types, root causes, and the impact of bugs in DL libraries and found that data and logic-related bugs are the most common bugs in DL libraries. Zhang et al. [15] studied DL application bugs built on top of TensorFlow and collected bugs from both Stackoverflow and Github projects. They find that fixing patterns and root causes correlate and suggest developers and researchers make automated bug detection approaches on top of root causes. Humbatova et al. [13] provided an extensive and comprehensive taxonomy of faults in DL libraries. They focused on TensorFlow, Keras, and PyTorch for their study. The notable difference between their work with existing studies is that they interviewed 20 researchers and practitioners to increase the reliability of their findings. There are a couple of differences between our work. First, our study merely focuses on Github commits while their study also mined data from Stack overflow posts. Second, they analyzed general bugs of DL libraries while we studied security vulnerabilities reported in CWE and CVE portals.

2) Studies on ML Implementation Bugs: The study conducted by Xiao et al. [52] covers security vulnerabilities in three popular DL libraries including TensorFlow, PyTorch, and Caffe. There are two major differences compared to our work. First, they examine the attack surfaces of DL libraries while we analyze the root cause of vulnerabilities in the back-end implementation of ML libraries. The attack surface of an ML library refers to the set of entry points that attackers can potentially exploit to gain unauthorized access, manipulate data, or disrupt ML operations, the root cause of vulnerabilities refers to the underlying programming errors, design flaws, or configuration issues that make the ML library vulnerable to attacks. Second, we present a comprehensive study of the specific vulnerabilities (security attacks) and identify 16 distinct types of vulnerabilities. While they only focus on four potential threats to DL libraries. Thung et al. [18] studied three popular Java-based ML libraries to characterize bugs related to the implementation of such tools. They investigated bug reports and bug repositories of the subject programs and extracted data from the JIRA issue tracking system, analyzed 500 bugs, and addressed the research questions. There are major differences compared to our empirical study. First, they focus on analyzing bugs in three Java-based ML libraries while we focus on seven ML and DL libraries that are developed in Python and C/C++. Second, our study focuses on vulnerability types, root causes, symptoms, and fixing patterns while they only focus on the frequency and severity of bugs as well as bug localization. Jia et al. [17] conducted an empirical study on implementation bugs of TensorFlow. More specifically, they targeted more than 36k Github projects that use TensorFlow and extract pull requests, bug reports, and code changes from the corresponding repositories to address the research questions. There are major differences compared to our work. First, they merely focus on the TensorFlow library while we focus on six more ML libraries other than TensorFlow that cover state-of-the-art DL models, classic ML models, and data analysis. Second, we provide 16 different vulnerability types based on CWE number while they merely focus on root causes, symptoms, and fixing patterns. The most related papers to our study are the studies conducted by Franco et al. [20] and Shen et al. [16]. Franco et al. [20] conducted the first study on characteristics of real-world numerical bugs of different numerical libraries, including NumPy, SciPy, LAPACK, GNU Scientific Library, and Elementa. They find that 32% of bugs in the studied libraries are related to *Numeric*. Our study complements their analysis in the sense that ours is more general since we study both numerical and ML libraries. Shen et al. [16] proposed an empirical study on DL compiler bugs by manually analyzing 595 bugs from Apache TVM, Facebook Glow, and Intel nGraph. The difference between this work and our study is that they focus on DL compiler bugs and provide guidelines for detecting and debugging them, while our study focuses on security vulnerabilities in ML libraries and characterizes their types, root causes, symptoms, and fixing patterns.

VII. Conclusion

This paper conducts the first empirical study to understand the characteristics of software security vulnerabilities of ML libraries. The primary motivation behind this study was twofold; characterize ML security vulnerabilities, and help ML developers design and develop vulnerability detection and debugging techniques to increase the quality and reliability of ML libraries. To achieve this goal, we manually analyzed 683 commits from seven widely used ML libraries, including TensorFlow, PyTorch, Scikit-learn, Mlpack, Scipy, Pandas, and Numpy. The outcome of this study is 16 vulnerability types, 20 root causes, 4 symptoms, 19 fixing patterns, and ultimately six findings. Based on these findings, we further provide a set of actionable guidelines to the ML community to design and develop software vulnerability detection and debugging techniques to increase the reliability and security of ML libraries.

Acknowledgments

The authors thank the anonymous reviewers for their feedback which helped improve this paper.

Availability

We make the dataset and source code of our experiments available at https://cse19922021.github.io/Deep-Learning-Security-Vulnerabilities/.

References

[1] G. Algan and I. Ulusoy, "Image classification with deep learning in the presence of noisy labels: A survey," *Knowledge-Based Systems*, vol. 215, p. 106771, 2021.

[2] F. Mahdisoltani, G. Berger, W. Gharbieh, D. Fleet, and R. Memisevic, "Fine-grained video classification and captioning," *arXiv preprint arXiv:1804.09235*, vol. 5, no. 6, 2018.

[3] R. Patgiri, "A taxonomy on big data: Survey," *arXiv preprint arXiv:1808.08474*, 2018.

[4] Y. Lv, B. Liu, J. Zhang, Y. Dai, A. Li, and T. Zhang, "Semi-supervised active salient object detection," *Pattern Recognition*, vol. 123, p. 108364, 2022.

[5] R. Simhambhatla, K. Okiah, S. Kuchkula, and R. Slater, "Self-driving cars: Evaluation of deep learning techniques for object detection in different driving conditions," *SMU Data Science Review*, vol. 2, no. 1, p. 23, 2019.

[6] S. Ramos, S. Gehrig, P. Pinggera, U. Franke, and C. Rother, "Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 1025–1032.

[7] R. Kulkarni, S. Dhavalikar, and S. Bangar, "Traffic light detection and recognition for self driving cars using deep learning," in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*. IEEE, 2018, pp. 1–4.

[8] S. Minaee and Z. Liu, "Automatic question-answering using a deep similarity neural network," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2017, pp. 923–927.

[9] R. G. Athreya, S. K. Bansal, A.-C. N. Ngomo, and R. Usbeck, "Template-based question answering using recursive neural networks," in *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*. IEEE, 2021, pp. 195–198.

[10] P. K. Roy, "Deep neural network to predict answer votes on community question answering sites," *Neural Processing Letters*, vol. 53, no. 2, pp. 1633–1646, 2021.

[11] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[12] J.-W. Hong, Y. Wang, and P. Lanz, "Why is artificial intelligence blamed more? analysis of faulting artificial intelligence for self-driving car accidents in experimental settings," *International Journal of Human–Computer Interaction*, vol. 36, no. 18, pp. 1768–1774, 2020.

[13] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1110–1121.

[14] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 510–520.

[15] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 129–140.

[16] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 968–980.

[17] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021.

[18] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 271–280.

[19] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 385–396.

[20] A. Di Franco, H. Guo, and C. Rubio-González, "A comprehensive study of real-world numerical bug characteristics," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 509–519.

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

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[24] M. L. Y. M. S. G. S. Z. R.R. Curtin, M. Edel, "mlpack 3: a fast, flexible c++ machine learning library," p. 726, 2018.

[25] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445. Austin, TX, 2010, pp. 51–56.

[26] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[27] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[28] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.

[29] W. Al-Kahla, A. S. Shatnawi, and E. Taqieddin, "A taxonomy of web security vulnerabilities," in *2021 12th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2021, pp. 424–429.

[30] P. Sharma and J. Singh, "Systematic literature review on software effort estimation using machine learning approaches," in *2017 International Conference on Next Generation Computing and Information Systems (ICNGCIS)*. IEEE, 2017, pp. 43–47.

[31] T. Aslam, "A taxonomy of security faults in the unix operating system," *Master's thesis, Purdue University*, vol. 199, no. 5, 1995.

[32] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[33] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 42–54.

[34] T. Hirsch and B. Hofer, "Root cause prediction based on bug reports," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 171–176.

[35] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[36] Facebook. (2013) Infer. [Online]. Available: https://fbinfer.com/

[37] D. A. Wheeler. (2013) Dlawfinder. [Online]. Available: http://dwheeler.com/flawfinder/

[38] A. Dunham. (2009) rough-auditing-tool-for-security. [Online]. Available: https://github.com/andrew-d/rough-auditing-tool-for-security

[39] D. Marjamäki. (2016) Cppcheck. [Online]. Available: https://cppcheck.sourceforge.io/

[40] G. inc. Errorprone. [Online]. Available: https://errorprone.info/

[41] SpotBugs. (2021) Spotbugs. [Online]. Available: .https://spotbugs.github.io/

[42] D. A. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 980–982.

[43] D. A. Tomassi and C. Rubio-González, "On the real-world effectiveness of static bug detectors at finding null pointer exceptions," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 292–303.

[44] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 317–328.

[45] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[46] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, "Identifying the characteristics of vulnerable code changes: An empirical study," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 257–268.

[47] M. Jimenez, M. Papadakis, and Y. Le Traon, "An empirical analysis of vulnerabilities in openssl and the linux kernel," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016, pp. 105–112.

[48] C. Q. Adamsen, A. Møller, S. Alimadadi, and F. Tip, "Practical ajax race detection for javascript web applications," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 38–48.

[49] E. Arteca, S. Harner, M. Pradel, and F. Tip, "Nessie: Automatically testing javascript apis with asynchronous callbacks." ICSE, 2022.

[50] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering*, 2021.

[51] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 103–112.

[52] Q. Xiao, K. Li, D. Zhang, and W. Xu, "Security risks in deep learning implementations," in *2018 IEEE Security and privacy workshops (SPW)*. IEEE, 2018, pp. 123–128.