

# Characterizing and Understanding Software Security Vulnerabilities in Machine Learning Libraries

Nima Shiri Harzevili  
York University  
nshiri@yorku.ca

Jiho Shin  
York University  
jihoshin@yorku.ca

Junjie Wang  
Chinese Academy of Sciences  
junjie@iscas.ac.cn

Song Wang  
York University  
wangsong@yorku.ca

## Abstract

The application of machine learning (ML) libraries has been 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 towards characterizing and understanding the security vulnerabilities of five well-known ML libraries, including TensorFlow, PyTorch, Scikit-learn, Pandas, and Numpy. To do so, we collected 596 security vulnerabilities to explore five major factors: 1) vulnerability types, 2) root causes, 3) symptoms, 4) fixing patterns, and 5) fixing efforts of security vulnerabilities in ML libraries. The findings of this study can help developers understand the characteristics of security vulnerabilities across different ML libraries. To make our finding actionable, we further developed DeepMut, an automated mutation testing tool, as a proof-of-concept application of our findings. DeepMut is designed to assess the adequacy of existing test suites of ML libraries against security-aware mutation operators extracted from the vulnerabilities studied in this work. We applied DeepMut on the TensorFlow kernel module and found more than 1k alive mutants not covered by the existing test suits, which have been confirmed by the development team of TensorFlow. The results demonstrate the usefulness of our findings.

## 1 Introduction

Nowadays, machine learning (ML) libraries have been frequently used in a wide variety of domains including but not limited to image classification [4, 22], big data analysis [27], pattern recognition [20], self-driving [17, 30, 33] and Natural Language Processing [6, 25, 31]. These ML libraries can be vulnerable to many attacks [21], and failures to detect the vulnerabilities in these libraries could cause catastrophic outcomes, such as car accidents [11].

In the past years, there have been multiple research studies to characterize ML bugs from end-users' context in

which bugs are mainly related to API usage of ML libraries [12, 13, 32, 39], or developers' context where the bugs are located inside components or core algorithms of ML libraries (implementation bugs) [8, 9, 14, 32, 35]. For example, Zhang et al. [39] focused on one typical deep learning (DL) library, i.e., TensorFlow, and studied DL application bugs built on top of TensorFlow. Islam et al. [8] 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 ML libraries? What are the root causes of security vulnerabilities in ML libraries? What symptoms do these security vulnerabilities have? Are there any fixing patterns for resolving these security vulnerabilities? And What are the efforts required to fix these security vulnerabilities?*

Understanding such characteristics of security vulnerabilities in 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 five significant factors: 1) vulnerability types, 2) root causes, 3) symptoms, 4) fixing patterns, and 5) fixing effort of security vulnerabilities in five well-known ML libraries including TensorFlow [2], Keras [1], PyTorch [26], Scikit-learn [28], Pandas [24], and Numpy [10]. For our study, we consider all available commits when we conduct this study on Sept. 1st, 2021, to collect security vulnerabilities in each ML library. We first searched commit messages with keywords that are related to vulnerabilities (details are in Section 2.1) to identify commits that fixed security vulnerabilities. As a result, 4K commits are collected. We then manually check each commit collected in the first step and identify and characterize vulnerabilities from it by

following systematic processes (details are in Section 2.2). In total, we obtained 596 unique security vulnerabilities from the studied five ML libraries. In this paper, we are to address the following research questions:

**RQ1: What types of vulnerabilities exist in ML libraries?** During the development and maintenance of ML libraries, developers often have to deal with various vulnerabilities of different types. To better understand vulnerabilities, this research question categorizes vulnerabilities and demonstrates their frequencies and distributions for each library. In this paper, we categorize vulnerability types based on Common Weakness Enumeration (CWE)<sup>1</sup>.

**RQ2: What are the root causes for vulnerabilities in ML libraries?** To understand the nature of ML vulnerabilities, it is critical to identify the root cause, which helps developers explore potential approaches to avoiding and fixing vulnerabilities. This research question examines the detailed root causes of vulnerabilities among the studied ML libraries.

**RQ3: What are the symptoms of vulnerabilities in ML libraries?** This research question categorizes the symptoms or effects of different vulnerabilities in ML libraries as understanding the symptoms can help developers assess the impact of vulnerabilities and triage them appropriately. In addition, the symptoms can help developers identify vulnerabilities quickly during software testing.

**RQ4: What are the fixing patterns for vulnerabilities in ML libraries?** Fixing patterns provide general solutions for resolving specific types of vulnerabilities. In this research question, we study the patches of each vulnerability to identify and analyze its fixing resolution. Common fixing resolutions across multiple vulnerabilities are grouped into different fixing patterns.

**RQ5: What are the efforts required for fixing vulnerabilities in ML libraries?** Fixing effort can help measure how much effort the development team has allocated to fix vulnerabilities. In this paper we are following existing studies [8, 18, 34] and use the line of code changed to measure the fixing effort of vulnerabilities.

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 provide a set of practical guidelines to help machine learning development teams to develop reliable and secure ML libraries.
- We develop DeepMut, an automated mutation testing tool for ML libraries as a proof-of-concept application of our findings. DeepMut is developed to evaluate the adequacy of the existing test suite of ML libraries against security-aware mutation operators extracted from the studied vulnerabilities in ML libraries.

- We have applied DeepMut on the TensorFlow kernel module and found more than 1k alive mutants that are not covered by the existing test suite of TensorFlow.
- We release the dataset and source code of our experiments to help other researchers replicate and extend our study<sup>2</sup>.

## 2 Methodology

### 2.1 Data Collection

This paper studies five widely used ML libraries, including TensorFlow, PyTorch, Scikit-learn, Pandas, and Numpy. The reason is that the studied ML libraries cover the current industrial machine learning practice and represent the critical aspects of machine learning developments. For example, TensorFlow is low-level, while PyTorch provides high-level APIs to hide the low-level details. Scikit-learn is a machine learning library with hundreds of APIs to build various machine learning models. Pandas and Numpy are two famous data analysis and visualization tools focusing on working with arrays and data frames. Thus, studying these libraries can help provide a comprehensive understanding of software vulnerabilities in ML libraries and further assist practitioners to build more reliable and secure ML libraries. We excluded some popular deep learning libraries, such as Caffe, Keras, and Theano, from our experiment subjects. The reason is that we cannot get sufficient historical security vulnerabilities from their repositories, i.e., we could only get 15 security-related commits from Keras’s GitHub repository. For each library, we consider all available commits when we conduct this study on Sept. 1st, 2021, to collect security vulnerabilities. Table 1 shows the statistics of our experiment projects. Our vulnerability collection process consists of the following two steps.

**Step 1: Vulnerability fixing commit collection.** We first extracted all the public CVEs of each experimental project available in the National Vulnerability Database (NVD) on Sept. 1st, 2021. We consider commits whose commit messages contain these CVEs as the fixing commits to these vulnerabilities by following existing studies [36, 37]. Note that, as reported in existing studies [29, 38], not all security vulnerabilities have CVE identifiers. For example, in our data collection process, we found that four of the five experimental subjects (i.e., PyTorch, Scikit-Learn, Pandas, Numpy) do not have CVEs so far. To cover all possible vulnerabilities, we used the heuristical approaches proposed by Zhou et al. [42], to identify the security fixing commits. Specifically, we follow their study and use their designed regular expression rules, including possible expressions and keywords related to security issues, to collect security vulnerability fixing commits. As a

<sup>1</sup><https://cwe.mitre.org/>

<sup>2</sup><https://cse19922021.github.io/Deep-Learning-Security-Vulnerabilities/>

Table 1: 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
Numpy	N.A	1,198	150	C/Python
<b>Overall</b>	36	4,152	596	-

result, we collected 4,152 fixing commits on the five studied projects (details are given in Table 1).

**Step 2: Vulnerability Identification.** Our heuristic approaches to vulnerability fixing commit collection might contain noise as the approach can introduce false positives [42]. In addition, an existing study showed that one security vulnerability could have multiple fixing commits, which need to be grouped for having a complete picture of the involved vulnerability [37]. Thus, to reduce noise and make our dataset more accurate, we further conduct a manual analysis to group possible fixing commits and identify unique security vulnerabilities on the data collected in step 1. In particular, the authors separately inspected each candidate vulnerability fixing commits to identify vulnerabilities, and they investigated inconsistent cases together to reach a consensus. Finally, we identified 596 unique vulnerabilities from the 4,152 fixing commits collected in Step 1.

## 2.2 Data Labeling

In our study, we analyzed each vulnerability from multiple aspects: 1) vulnerability type, 2) root cause, 3) symptom, 4) fixing pattern, and 5) fixing effort. Please note that some existing studies on analyzing general software bugs in machine learning libraries have also provided taxonomies for these aspects [13, 32]. In this work, we did not adopt corresponding taxonomies from these studies. The reason is that existing studies merely focus on general software bug characteristics of ML libraries either from end-user or developers’ perspectives. In other words, they do not provide categorizations for vulnerabilities of 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 our labeling process, two authors work together to review each identified vulnerability. In particular, they check the related artifact of this vulnerability, including fixing commits, developer discussion, and pull requests to carefully understand the vulnerability and provide the following information for each vulnerability: 1) vulnerability type, 2) root cause, 3) symptom, 4) fixing pattern, and 5) fixing effort. The two authors discussed the disagreements together during the labeling process until all information was extracted consistently.

## 3 Result Analysis

In this section, we present and discuss our analysis results to address the five research questions we asked in Section 1.

### 3.1 RQ1: Vulnerability Types

The taxonomy of vulnerability types is shown in Figure 1. It is organized into five high-level categories (i.e., **Memory**, **Resource**, **Numeric**, **Buffer**, **Concurrency**) and involves more than 19 different CWEs covered by 567 (94.6%) of the 596 vulnerabilities. The remaining 32 (5.34%) vulnerabilities appear infrequently and do not belong to any particular vulnerabilities, and are included in **Others** category.

**Numeric.** Vulnerabilities in this category mostly deal with improper calculation or conversion of numbers, accounting for 184 (30.8%) 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)*.

**Memory.** This category contains vulnerabilities consuming GPU memory abnormality, accounting for 179 (30%) 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)*.

**Buffer.** This type of vulnerability corresponds to the handling of memory buffers within a software system, accounting for 89 (14.9%) 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)*.

**Resource.** Vulnerabilities in this category correspond to resource initialization or validation issues, accounting for 66 (11%) of the vulnerabilities. It consists of three types of CWEs: 1) *Use of Uninitialized Resource (CWE-908)*, 2) *Improper Input Validation (CWE-20)*, and 3) *File Descriptor Leak (CWE-403)*.

**Concurrency.** Vulnerabilities in this category relate to concurrent access of resources and their locking applied by multiple threads, accounting for 46 (11.5%) vulnerabilities. It consists of two types of CWEs: 1) *Race Condition (CWE-362)* and 2) *Deadlock (CWE-833)*.

Figure 2 shows the distribution of each type of vulnerability in the studied five libraries. As we can see, **Numeric** and **Memory** are the two dominating types across all libraries. Specifically, **Numeric** is the most common type of vulnerability among libraries except Numpy library, where *Memory Leak* is the most common vulnerability. We demonstrate distribution of subcategories for **Numeric** and **Memory** in Table 2 and Table 3. As shown in Table 2, *Integer Overflow (CWE-190)* is the most common vulnerability type among other

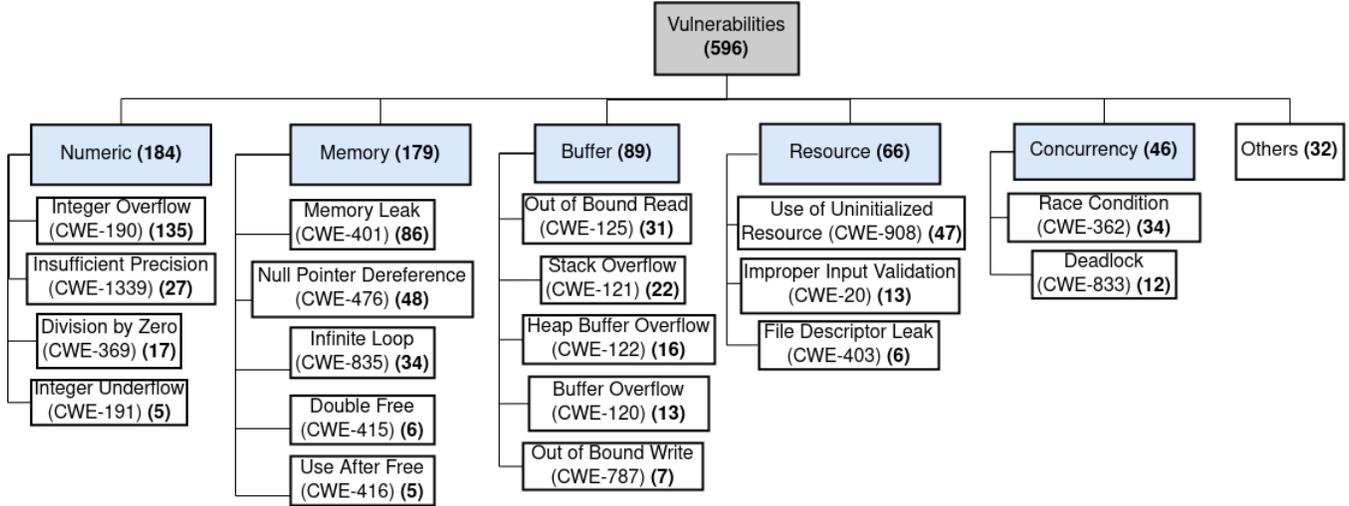


Figure 1: The taxonomy of vulnerability types studied in this work.

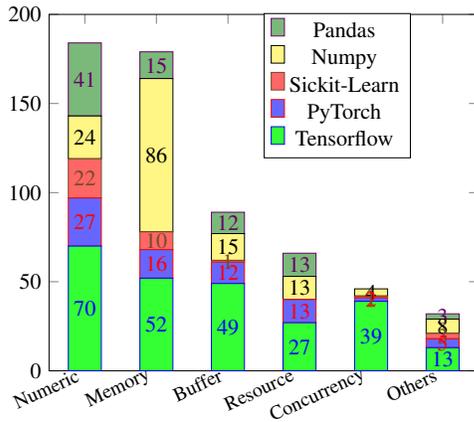


Figure 2: The distribution of software vulnerabilities in different ML libraries.

subcategories for all libraries. This states that *Integer Overflow* in the studied ML libraries is critical, and developers need to pay more attention to this type.

**Finding 1:** Vulnerabilities in the categories of **Numeric** and **Memory** are most frequent vulnerabilities across all libraries, and *Integer Overflow* is the most common vulnerability in the five ML libraries.

### 3.2 RQ2: Root Causes

The taxonomy of root causes of studied vulnerabilities in ML libraries is shown in Figure 3, which is organized into five high-level categories including **Data Type Errors**, **Memory**

Table 2: Subcategories of **Numeric**.

Library	Integer Overflow	Insufficient Precision	Division by Zero	Integer Underflow
Tensorflow	63	4	2	1
PyTorch	18	6	3	0
Sickit-learn	8	4	7	3
Pandas	34	3	3	1
Numpy	12	10	2	0
<b>Sum</b>	<b>135</b>	<b>27</b>	<b>17</b>	<b>5</b>

Table 3: Subcategories of **Memory**.

Library	Memory Leak	Null Pointer Dereference	Infinite Loop	Double Free	Use After Free
Tensorflow	6	25	19	1	1
PyTorch	2	7	5	2	0
Sickit-learn	8	0	2	0	0
Pandas	5	6	3	1	0
Numpy	65	10	5	2	4
<b>Sum</b>	<b>86</b>	<b>48</b>	<b>34</b>	<b>6</b>	<b>5</b>

**Errors, API Errors, Business Logic Errors, and Concurrency Errors** covered by 558 (91.9%) of the 596 vulnerabilities. The remaining 48 (5.1%) root causes have no clear indication about their types, and hence we group them in **Others** category.

**Data Type Errors.** Root causes in this category mostly deal with range or precision issues of conventional data types defined by developers accounting for 213 (35.7%) of vulnerabilities. Subcategories include 1) *Numerical Precision Error*: When developers define variables or tensors with a limited or large range, 2) *Tensor Property Issue*: When a thread or a program maintains tensors inappropriately, 3) *Using Improper Data Type*: When developers have confusion about using data types, e.g., using int32 instead of uint32, 4) *Incorrect Type Conversion*: When a developer incorrectly convert data types together, e.g., implicit type conversion of float to double.

**Memory Errors.** Root causes in this category mainly deal with memory-related vulnerabilities accounting for 29.5% of vulnerabilities. The subcategories of this root cause are includ-

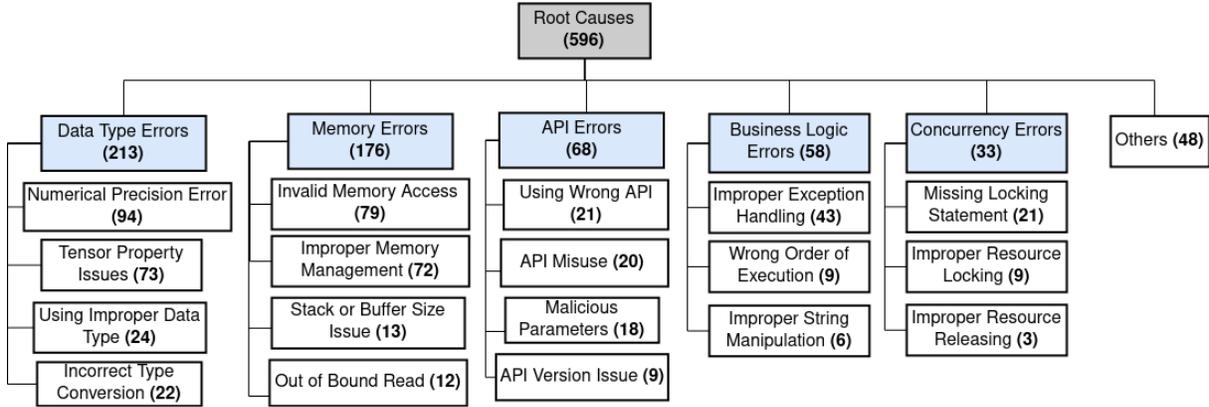


Figure 3: Taxonomy of root causes in ML libraries.

ing 1) *Invalid Memory Access*: When processes try to access memory locations filled with null values and already been deleted or freed, 2) *Improper Memory Management*: When a developer has confusion in memory management, either misuse memory release statement or forget to release memory after its lifetime, 3) *Out of Bound Read*: When processes read information from other memory locations, and 4) *Stack or Buffer Size Issue*: When developers define the stacks or buffers with inappropriate sizes.

**API Errors.** This root cause category is due to inconsistencies in updating or using APIs accounting for 58 (11.4%) 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, mistakenly using optional parameters, etc., 2) *Using Wrong API*: When developers mistakenly use improper APIs, 3) *Malicious Parameters*: When developers pass malicious or invalid parameters to API calls which are exploitable by attackers. Attackers can exploit these parameters by crafting particular inputs to take control of the software system, and 4) *API Version Issue*: When developers mistakenly use either wrong versions of APIs or outdated ones.

**Business Logic Errors.** This root cause accounts for 58 (9.7%) 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) *Wrong Order of Execution*: When a set of steps or components were inappropriately executed, 3) *Improper String Manipulation*: When developers parse string variables or absolute and relative addresses incorrectly.

**Concurrency Errors.** This root cause 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 33 (5.5%) vulnerabilities. Subcategories are 1) *Missing Locking Statement*: When developers forget to lock resources which mostly result in race condition or deadlock errors, 2) *Improper Resource*

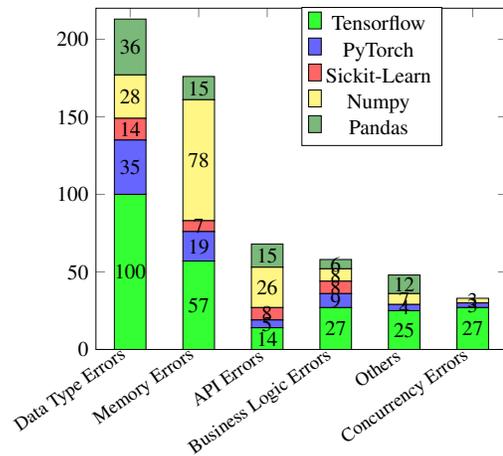


Figure 4: Distribution of root causes across libraries

*Locking*: When developers use locking statements improperly on program resources, 3) *Improper Resource Releasing*: When developers release locked resource inappropriately, which can result in deadlock or race condition errors.

Figure 4 shows the distribution of root causes of vulnerabilities in different libraries. As can be seen, **Data Type Errors** is the most common root cause of vulnerabilities across the studied ML libraries. Table 4 further shows the distribution of subcategories of **Data Type Errors** across the studied ML libraries. As we can see, *Numerical Precision Error* is the major subcategory because the studied ML libraries mostly rely on tensor level and array level computations. The computations involve quantization during training which means millions of parameters are multiplying or adding together in integer or float types to make models as smaller as possible. *Tensor Property Issue* is the second most common root cause of vulnerabilities. The reason is that optimization and quantization operations are mostly done with tensors.

**Memory Errors** is the second most common root cause of vulnerabilities across the studied ML libraries. Table 5

Table 4: Subcategories of **Data Type Errors**

Library	NPE <sup>1</sup>	TPI <sup>2</sup>	UDP <sup>3</sup>	ITC <sup>4</sup>
Tensorflow	34	47	13	6
PyTorch	17	11	5	2
Sickit-learn	12	0	1	1
Pandas	21	2	4	9
Numpy	10	13	1	4
<b>Sum</b>	94	73	24	22

<sup>1</sup> Numerical Precision Error | <sup>2</sup> Tensor Property Issues | <sup>3</sup> Using Improper Data Type | <sup>4</sup> Incorrect Type Conversion

Table 5: Subcategories of **Memory Errors**

Library	IMA <sup>1</sup>	IMM <sup>2</sup>	SBSI <sup>3</sup>	OOBR <sup>4</sup>
Tensorflow	41	4	7	5
PyTorch	17	1	1	0
Sickit-learn	0	6	0	1
Pandas	8	3	1	3
Numpy	13	58	4	3
<b>Sum</b>	79	72	13	12

<sup>1</sup>Invalid Memory Access | <sup>2</sup>Improper Memory Management | <sup>3</sup>Stack or Buffer Size Issue. <sup>4</sup>Out of Bound Read|

further shows the distribution of **Memory Errors**. As you can see, **Invalid Memory Access** is the major root cause of memory-related vulnerabilities. An invalid memory is a memory that is undefined, uninitialized, deleted, containing null values, corrupted values, erased, etc, **Improper Memory Management** is the second most common subcategory of **Memory Errors**. **Improper Memory Management** is the major root cause of the Numpy library, which is written in C language and memory management is the responsibility of developers. Developers of Numpy often forget to release the allocated memory address when its effective lifetime is finished.

**Finding 2: Data Type Errors and Memory Errors** are the most common types of root cause of vulnerabilities accounting for 64.2% of vulnerabilities in the studied ML libraries respectively. *Numerical Precision Error* is the dominating subcategory.

**API Errors** are the third most common root causes of vulnerabilities in the studied ML libraries. A more detailed distribution of subcategories are shown in Table 6. As you can see, **Using Wrong API** is the most common subcategory. The second common subcategory is **API Misuse** where developers have a hard time using APIs, e.g., passing wrong parameters, lack of using optional parameters, and improperly using optional parameters. **Malicious Parameters** are also common where developers give unsafe inputs to API calls which attackers can exploit. Sometimes developers use outdated or invalid APIs, which are the root cause of vulner-

Table 6: Subcategories of **API Errors**

Library	UWA <sup>1</sup>	AM <sup>2</sup>	MP <sup>3</sup>	AVI <sup>3</sup>
Tensorflow	3	3	3	5
PyTorch	2	1	1	1
Sickit-learn	2	3	3	0
Pandas	3	4	7	1
Numpy	11	9	4	2
<b>Sum</b>	21	20	18	9
<b>Sum</b>	21	20	18	9

<sup>1</sup> Using Wrong API | <sup>2</sup> API Misuse | <sup>3</sup> Malicious Parameters | <sup>4</sup> API Version Issue

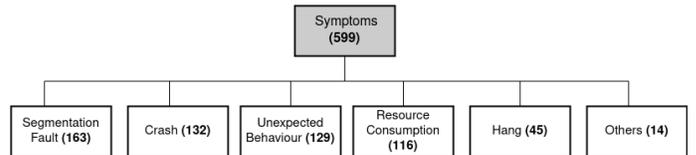


Figure 5: Taxonomy of symptoms in the studied ML libraries

abilities in the studied ML libraries. We categorize them as **API Version Issue**.

**API Errors** differ noticeably compared to traditional software systems. According to a current study on **API Errors** conducted by Amann et al. [5], missing and redundant API calls are the most frequent errors. At the same time, these are the least errors 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: API Errors** in the studied ML libraries cover more corner cases that are exploitable by attackers compared to traditional software systems.

### 3.3 RQ3: Symptoms

The taxonomy of symptoms of studied vulnerabilities in ML libraries is shown in Figure 5, which is organized into six categories including **Segmentation Fault**, **Crash**, and **Unexpected Behaviour**, **Resource Consumption**, and **Hang**, covered by 582 (97.6%) vulnerabilities. The remaining 14 (2.3%) symptoms have no clear indication about their outcome, and hence we group them in **Others** category.

**Unexpected Behavior:** If the library is producing results or behaving that is not expected. For example, in this **Integer Overflow** vulnerability from sklearn library<sup>3</sup> where  $pk * qk$  returns *inf* instead of *float* because of exceeding int32 bits limits during multiplication.

<sup>3</sup><https://github.com/scikit-learn/scikit-learn/commit/622f912095308733ddfe572a619b1574b9da335e>

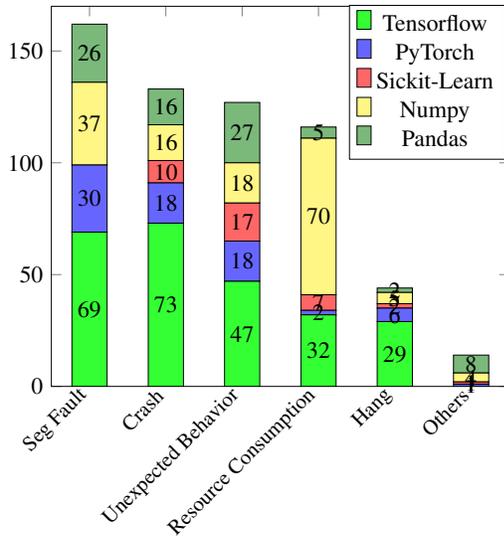


Figure 6: Distribution of symptoms across different libraries.

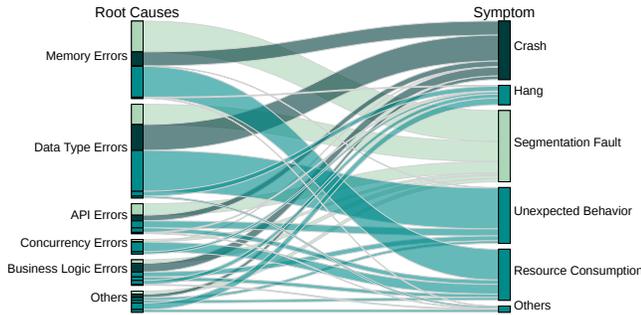


Figure 7: Mapping of root causes to symptoms.

Other symptoms includes **Segmentation Fault**: When a program outputs *core dumped* in the output which is the symptom for segmentation fault<sup>4</sup>, **Resource Consumption**: Exhaustion of available resource, e.g., increasing memory usage because of uncontrolled or improper allocation of main memory. **Hang**: This means that a program keeps running for a long period without termination or responding, and **Crash**: When a program or process terminates unexpectedly at runtime.

Figure 6 demonstrates the distribution of symptoms across different libraries. As you can see, the most frequent symptom is **Segmentation Fault** accounting for 27.3% of vulnerabilities. We also draw a mapping from root causes to symptoms to interpret what is the outcome of vulnerabilities as shown in Figure 7. It is observable from Figure 7 that vulnerabilities caused by **Memory Errors**, **Data Type Errors**, and **API Errors** often have **Segmentation Fault** as their symptom. Also, we can see the same pattern for **Crash** symptoms. Often

<sup>4</sup>[https://stackoverflow.com/questions/49092527/illegal-instructioncore-dumped-tensorflow](https://stackoverflow.com/questions/49092527/illegal-instruction-core-dumped-tensorflow)

developers can extract descriptive information from **Segmentation Fault** and **Crash**. One possible usage scenario of the extracted information is that they can parse stack traces of failed test suits to analyze the exact root causes of the vulnerability and how to locate them. Also, Developers do not need to develop test oracles in order to understand **Segmentation Fault** and **Crash** symptoms.

**Finding 4: Segmentation Fault and Crash** are the most common symptoms of vulnerabilities accounting for 27.3% and 22.1% of vulnerabilities respectively. These symptoms can help developers of the studied ML libraries to understand and locate the root cause of vulnerabilities with their descriptive information.

**Unexpected Behaviour** is the third most common symptom of vulnerabilities accounting for 21.3% of vulnerabilities. As shown in Figure 7, often **Unexpected Behaviour** is the symptom of vulnerabilities caused by **Data Type Errors**. Hence, developers of the studied ML libraries need to equip the existing test suite with test oracles to understand that ML libraries' components are working as they are expected to do so.

**Finding 5: Unexpected Behaviour** is the third most common symptom of vulnerabilities accounting for 21.3% of vulnerabilities. Its prevalence might suggest that developers need to develop test oracles to understand that the studied ML libraries meet their requirements defined by either end-users or developers.

### 3.4 RQ4: Fixing Patterns

The taxonomy of fixing patterns of ML vulnerabilities is shown in Figure 8, which is organized into six high-level categories including **Add Checkers**, **Resolve Data Type Errors**, **Resolve Memory Errors**, **Resolve API Errors**, **Resolve Concurrency Errors**, and **Modify Business Logic Errors** covered by 535 (89.7%) of the 596 vulnerabilities. The remaining 61 (10.2%) fixing patterns have no clear indication about their types, and hence are included in the **Others** category.

**Add Checkers**. Fixing patterns in this category are mainly about the addition of either library-specific checkers or conventional checkers to fix vulnerabilities, which cover 176 (29.5%) vulnerabilities. Subcategories are 1) *Add Checker for Tensors Property*: This is the most common fixing pattern where developers use if conditions or library-specific checkers to check tensor or arrays properties, e.g., shapes, ranks, values, or elements.

2) *Add Checker for Overflow*: This fixing pattern is mainly used to fix overflow vulnerabilities where developers either

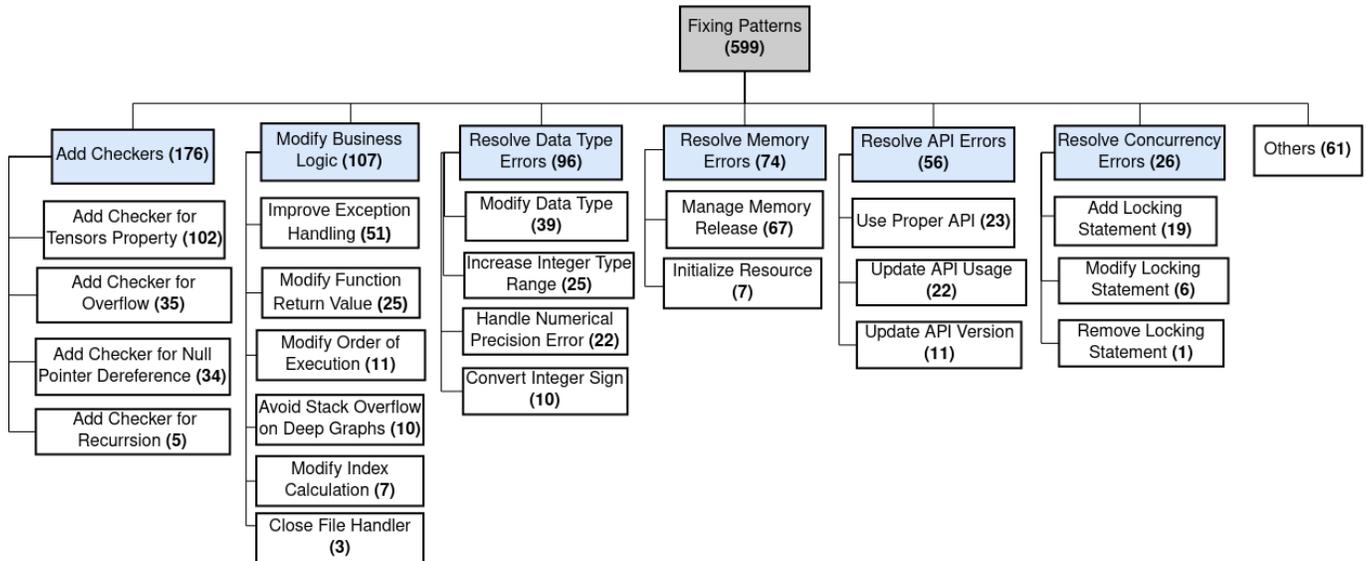


Figure 8: Taxonomy of fixing patterns in ML libraries.

add if modules and library-specific checkers or add functions for mathematical operations instead of using explicit mathematical operators, 3) *Adding Checker for Null Pointer Dereference*: Developers often add checkers either using if conditions or library-specific checkers to fix null pointer dereferences, and 4) *Adding Checker for Recursion*: Sometimes an infinite loop occurs due to endless recursion calls. So, developers need to detect recursions that consume stack space, and checkers are added to fix these types of vulnerabilities.

**Modify Business Logic:** This type of fixing pattern is related chiefly to modifying the existing control flows, functions, or classes to fix vulnerabilities that have incorrect logic. Subcategories are 1) *Improved Exception Handling*: When the program crashes, it is necessary to record the error or warning message to help developers with debugging. This pattern adds missing error reporting or modifying existing ones that are defective, 2) *Modifying Function Return Value*: This pattern mainly changes function return values to avoid potential mismatches in the data flow of a program for fixing vulnerabilities, 3) *Modify Order of Execution*: Developers change the location of semantically related statements to fix vulnerabilities, 4) *Avoid Stack Overflow on Deep Graphs*: This pattern is used when developers try to prevent stack overflow caused by small stack size or deep computation graphs created in runtime, 5) *Modify Index Calculation*: This pattern is used to fix vulnerabilities related to incorrect indices of data collection such as arrays or tensors, and 6) *Close File Handler to Prevent File Leak*: It is used to fix file descriptor leak vulnerability.

**Resolve Data Type Errors:** Fixing patterns in this category focus on resolving vulnerabilities related to data types, which cover 16.1 % of the studied ML vulnerabilities. Subcategories include: 1) *Modify Data Type*: is used to fix vulnerabilities involving incorrect data type defined and used, 2) *Increase*

*Integer Type Range*: is used to fix vulnerabilities that are caused by the limited range of integer types for preventing integer overflow or integer truncation, e.g., using int64 instead of int32, 3) *Handle Numerical Precision*: is used to resolve data type precision issues, e.g., normalization of matrix values during float 16 bits model training, 4) *Convert Integer Sign*: is used to convert data type signs to prevent integer overflow or underflow, e.g., using size\_t instead of int32.

**Resolve Memory Errors.** Fixing patterns in this category relate to memory management efforts, which can help fix 74 (12.4%) 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, variables, or data types to fix vulnerabilities.

**Resolve API Errors:** Fixing patterns in this category are mainly used to fix vulnerabilities introduced by inappropriate API usages, which help fix 56 (9.3 %) of vulnerabilities studied in this paper. The detailed subcategories are 1) *Using Proper API*, 2) *Update API Usage*, and 3) *Update API Version*.

**Resolve Concurrency Errors.** Fixing patterns in this category are used to fix vulnerabilities related to concurrency issues resulting in deadlock or race condition errors. Subcategories include 1) *Add Locking Mechanism*, 2) *Modify Locking Mechanism*, and 3) *Remove Locking Mechanism*.

Figure 10 shows the distribution of fixing patterns across different libraries. As can be seen, **Adding Checkers** is the most common fixing pattern in ML libraries, accounting for 96, 24, 6, 32, 18 vulnerabilities of TensorFlow, PyTorch, Scikit-Learn, Pandas, and Numpy, respectively. In total, 29.5% of vulnerabilities can be fixed by this pattern. We further show the breakdown of **Adding Checkers** regarding the distribu-

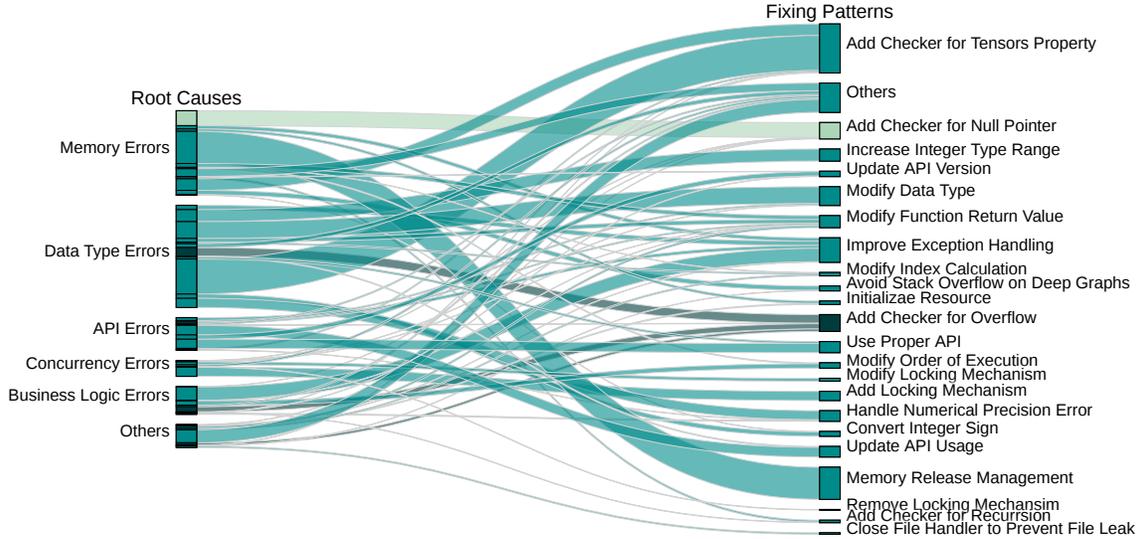


Figure 9: Mapping of root causes to fixing patterns.

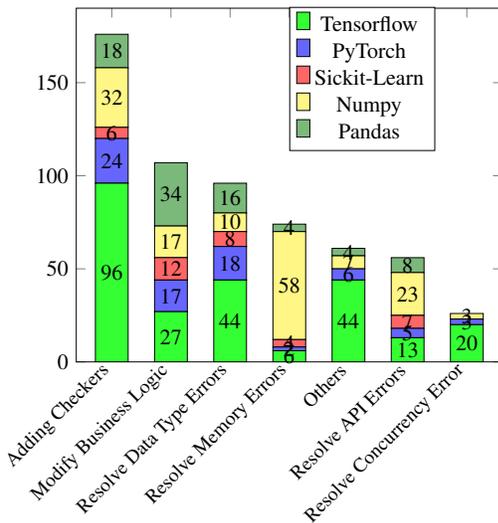


Figure 10: Distribution of fixing patterns across different libraries.

tions of subcategories in Table 7. As we can see, *Adding Checker for Tensor Property* is the principal subcategory that covers 102 vulnerabilities.

**Finding 6: Adding Checkers** is the most common fixing pattern across the studied ML libraries accounting for 17.9% of vulnerabilities.

Figure 9 illustrates the mapping of root causes to their corresponding fixing patterns. As shown in the figure, **Memory Errors** have three common fixing patterns including **Improper Memory Management**, and **Add Checker for Ten-**

Table 7: Subcategories of **Add Checkers**

	ACTP <sup>1</sup>	ACFO <sup>2</sup>	ACNP <sup>3</sup>	ACR <sup>4</sup>
Tensorflow	67	13	16	0
PyTorch	13	7	3	1
Sickit-learn	1	4	1	0
Pandas	7	10	1	0
Numpy	14	1	13	4
Sum	102	35	34	5

<sup>1</sup> Add Checker for Tensors Property | <sup>2</sup> Add Checker for Overflow | <sup>3</sup> Add Checker for Null Pointer Dereference | <sup>4</sup> Add Checker for Recursion

**sor Property, Add Checker for Null Pointer.** The major fixing pattern for **Memory Errors** is **Memory Release Management** where developers use memory management APIs to free allocated memories after their effective lifetime. This pattern is mostly used by the Numpy library community which is developed in C language and developers should take care of memory management manually. This **Memory Leak** vulnerability<sup>5</sup> perfectly explains how developers use **Memory Release Management** pattern to resolve the problem. In this vulnerability, the *slice* object is not decref'd, hence the developer calls *Py\_DECREF()* api with *slice* as the parameter to fix the **Memory Leak** problem.

The second fixing pattern which resolves **Memory Errors** is **Add Checker for Tensor Property**. Often, tensor properties can be problematic if developers do not employ appropriate checkers, either project-specific or general checkers. This vulnerability is a good example of how developers use checkers to prevent **Infinite Loop**. In this case, **Infinite Loop** occurs because *num\_cols* exceeds  $2^{31}-1$  which is the max-

<sup>5</sup><https://github.com/numpy/numpy/commit/4e19f408de900f958441af4ec8a458f5ce6473eb>

imum value an int32 bits variable can take. The developer uses two checkers of **OP\_REQUIRES** kind to guard against **Infinite Loop** via checking the dimensions of the input data must be less than or equal to  $2^{31-1}$ .

Figure 9 further shows that **Add Checker for Tensor Property** is the major fixing pattern to fix **Data Type Errors**. Often lack of checking tensor properties is the root cause of **Data Type Errors**, e.g. **Integer Overflow**. In this example<sup>6</sup> which is **Integer Overflow**, there is no checker on `data[axis]` to make sure it does not exceed in32 bits range limits. The developer overcomes the problem by adding a checker of **TF\_LITE\_ENSURE** on line 76 of `tensorflow/lite/kernels-concatenation.cc`.

**Finding 7: Lack of Checking Tensor Property** is a common cause of **Data Type Errors** and **Memory Errors** in the studied ML libraries. Developers can overcome the vulnerabilities by using **Add Checker for Tensor Property** as the fixing pattern.

### 3.5 RQ5: Fixing Effort

The taxonomy of fixing effort of the studied ML vulnerabilities is shown in Figure 11, which is organized into four categories. We adopted the categories from [8, 18, 34] to show the scales of fixing effort: 1) **Micro repair**: 0-50 added or deleted lines, 2) **Small repair**: more than 50 added or deleted lines, 3) **Medium repair**: 50-200 added or deleted lines, and 4) **Large repair**: more than 200 added or deleted lines. As shown in Figure 11, 73.9% of vulnerabilities can be fixed by micro and small fixing efforts. Also, Figure 12 further shows the distribution of fixing scales across different root causes. As we can see from the figure, the distributions of fixing effort of vulnerabilities of different root causes do not have dramatic difference, which may suggest the root cause of a vulnerability does not affect its fixing effort.

**Finding 8: Most (73.9%) vulnerabilities** in the studied ML libraries can be fixed in small scales. Fixing effort of a vulnerability is not related to its root cause.

## 4 Implications

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

**Avoid Data Type Errors.** According to our root cause analysis and finding 2 in Section 3.2, we conclude that **Data Type**

<sup>6</sup><https://github.com/tensorflow/tensorflow/commit/4253f96a58486ffe84b61c0415bb234a4632ee73>

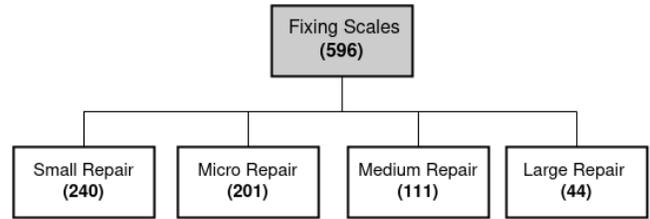


Figure 11: Taxonomy of fixing effort in the studied ML libraries.

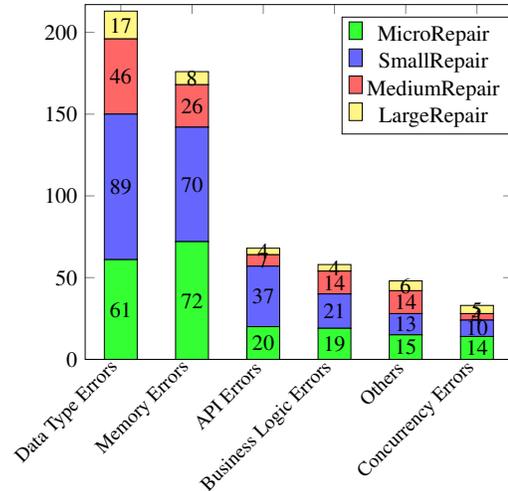


Figure 12: Distribution of fixing effort regarding different root causes.

**Errors** are the most common root cause of vulnerabilities. As a result, we suggest developers pay attention to 1) the range of integer variables they define, e.g., developers should define int64 bits instead of int32 bits to prevent overflow or integer truncation, 2) developers should pay attention to type conversions as it is one of the main reasons for integer overflow, and 3) developers should use checkers (either library-specific, e.g., **OP\_REQUIRES**, or if modules) to make sure that there are no vulnerabilities related to tensor properties, i.e., shapes, ranks, values, or elements.

**Always Initialize Variables.** Our analysis shows that 7.8% of vulnerabilities are due to initialization issues, e.g., lack of initializing tensors or variables. It is always better to initialize variables or any resource during the development of ML tasks.

**Apply Dynamic Analysis Tools to Avoid Memory Related Vulnerabilities.** We find that **Resource Consumption** is the fourth most common symptom of vulnerabilities accounting for 19.4 of vulnerabilities. Often, understanding whether libraries are suffering from **Resource Consumption** is challenging since test suits can not profile resource usage. Moreover, we find that **Memory Errors** is the second most root cause of vulnerabilities in ML libraries. The main reason for **Memory Errors** is memory release management, where

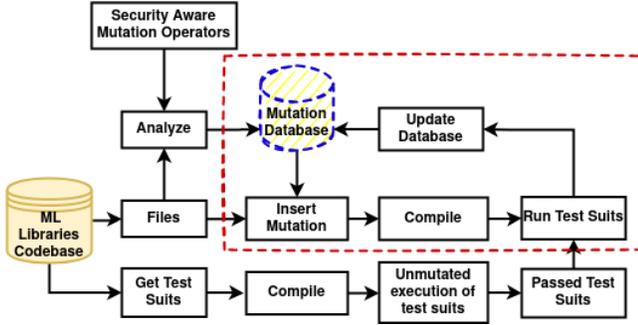


Figure 13: Architecture of DeepMut.

developers forget to release the allocated memories. Hence, we strongly suggest developers of studied ML libraries use dynamic checkers to detect and avoid memory-related vulnerabilities by applying memory checkers, e.g., Valgrind<sup>7</sup>. We have manually checked that Valgrind can detect most half of the **Memory Errors** related vulnerabilities, which suggests the importance of using dynamic analysis tools to avoid memory-related vulnerabilities.

**Use API Usage Checker.** As shown in Figure 3, **API Errors** is also a common root cause for vulnerabilities in ML libraries; API-related issues mainly cause vulnerabilities whose root causes fall in this category during the development of ML libraries. This root cause is decomposed into different subcategories as shown in Table 6. As we can see, *API Misuse*, *Using Wrong API*, and *API Version Issue* cover 50 out of the 68 **API Errors** involved vulnerabilities. We have checked that most of these types of API usage issues can be avoided by using API usage checkers [40, 41], thus developers are suggested to apply these API misuse detectors when working on ML development tasks for avoiding **API Errors** related vulnerabilities. The third most common API inconsistency is the API Version Issue, where developers mostly use improper versions of APIs, causing different vulnerabilities. Hence, there is a need to develop version control (checking) tools to assist developers in using up-to-date APIs.

## 5 Actionable Applications of Our Findings

We believe the findings of our study can be used to improve software vulnerability development tasks, such as detecting similar unknown vulnerabilities based on the studied vulnerabilities in this work, categorizing newly reported vulnerabilities, and improving security vulnerability testing via security mutation analysis.

To make our finding actionable, we take security mutation analysis as an example and develop DeepMut, an automated mutation testing tool for ML libraries, as a proof-of-concept application of our findings. DeepMut is designed to evaluate

```

79   BCast bcast(BCast::FromShape(input_shape), BCast::FromShape(output_shape),
80             /*fewer_dims_optimization=*/true);
81   OP_REQUIRES(ctx, bcast.IsValid(),
82             errors::InvalidArgument(
83               "Incompatible shapes: ", input_shape.DebugString(), " vs. ",
84               output_shape.DebugString());
85   OP_REQUIRES(ctx, BCast::ToShape(bcast.output_shape()) == output_shape,
86             errors::InvalidArgument("Unable to broadcast tensor of shape ",
87                                     input_shape, " to tensor of shape ",
88                                     output_shape));
  
```

Figure 14: An example of alive mutant found in Tensorflow kernel module.

the adequacy of the existing test suite of ML libraries against security-aware mutation operators extracted from the vulnerabilities studied in this work. Specifically, we use all the **Tensor property Issues** related vulnerabilities from TensorFlow as an example to extract mutation operators for DeepMut.

Figure 13 shows the overall architecture of DeepMut, which has the following steps to find alive mutants. First, it gets all source files from the target ML library then performs an initial analysis to extract potential statements that match with the extracted mutation operators. The potential statements are stored in the mutation database. The mutation loop starts iterating over the database and inserts each mutant into the target source code. Once the insertion is finished, the compilation process begins where the library under test is compiled. Subsequently, all test suites run against inserted mutants. To determine whether the mutant is killed or not, we perform a simple analysis on the stack trace of running test suites and update the database. This process continues until all mutants in the database are executed.

We performed DeepMut on *TensorFlow/core/kernel module* module. We ran DeepMut on branch v2.7.0<sup>8</sup>, one of the active branches of TensorFlow. It took around one week to get the initial results. In total, DeepMut generates more than 3k mutants, and among them 1.2K are alive.

Here, we further show an example of an alive mutant found by DeepMut from *broadcast\_to\_ops.cc* file (from TensorFlow kernel module) illustrated in Figure 14 that is not covered by the TensorFlow kernel’s test suite. As you can see, a checker at lines 81-84 is responsible for checking whether input and output shapes are compatible with each other. If the shapes violate the constraints defined in the checker, a run time error will be generated. First, DeepMut analyzes *broadcast\_to\_ops.cc* to search for potential mutation locations in this file and applicable mutation operators, as a result, DeepMut finds an applicable operator (i.e., eliminating *OP\_REQUIRES* and *OP\_REQUIRES\_OK* from source code) for the fixing patch of existing vulnerabilities in TensorFlow. To apply the operator, DeepMut removes lines 81-84 and compiles TensorFlow. Once the compilation is done, DeepMut runs all test suites of kernel modules and finds that none of the existing tests can detect the deleted lines.

<sup>7</sup><https://valgrind.org/>

<sup>8</sup><https://github.com/TensorFlow/TensorFlow/tree/r2.7>

We have reported these live mutants to the security team of TensorFlow. They confirmed that these scenarios are not covered by the existing test suite and agree that DeepMut is useful to help design security test cases. In this paper, we apply DeepMut on the TensorFlow core kernel module. However, DeepMut is extensible to more components or more ML libraries.

## 6 Threats to validity

**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 Ph.D. students have reviewed the collected commits in multiple rounds. The students discuss any possible disagreement after each round 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 five different ML libraries; two are very famous and widely used DL libraries, including TensorFlow and PyTorch; one of them is Scikit-learn which is a renowned classical ML library which often is used beside DL libraries. We also collected data from two well-known data analytics and visualization tools, including Pandas and Numpy. The reason behind this diverse data collection is to generalize our findings to wide domains and increase the reliability of findings. To augment our classification process and make them more accurate, besides reviewed commits, we also reviewed issues and merged pull requests linked to the parent commits. Please see Section 2.1 for further details.

## 7 Related Work

### 7.1 Studies on General Vulnerabilities

This section surveys the existing studies on analyzing the characteristics of vulnerabilities in general software projects.

#### 7.1.1 Vulnerabilities in General Software Systems

There are many efforts to characterize software security vulnerabilities in traditional software systems [7, 16, 34].

Jimenez et al. [16] analyzed characteristics of vulnerabilities of Linux kernel and OpenSSL. They collected 2k vulnerable git commits that are 1) reported in CVE, 2) have vulnerable keywords, and 3) have a CVE number in their message and title. They find 20 frequent types of vulnerabilities; among them, CWE-200 and CWE-119 are the dominant ones for Linux and CWE-119, CWE-399, and CWE-362, for OpenSSL. There are two significant differences with our analysis; 1) Based on their findings, **Numeric** and **Memory** are not major common vulnerabilities in general software

systems, 2) Unlike studied ML libraries, general software systems require complex efforts to fix vulnerabilities.

Tan et al. [34] 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. [7] analyzed code review requests from 10 software projects to identify vulnerable code changes. They developed a tool called *Gerrit-Miner* that mines code reviews from the Gerrit code review portal that is publicly available. They mined more than 260k reviews and analyzed 1k reviews thoroughly for the analysis. 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.

#### 7.1.2 Vulnerabilities in Software Ecosystems

Software ecosystems are vital in modern software development as they provide reusable packages to developers and increase development speed. Two notable ecosystems are *npm*<sup>9</sup> that supports *Node.js* packages and *PyPi*<sup>10</sup> that supports Python packages. Alfel et al. [3] conducted a study to characterise vulnerabilities in *PyPi*. They focused mostly on how long it takes to find and fix vulnerabilities in projects in *PyPi*. They found that *Cross-Site-Scripting (XSS)* and *Denial of Service (DoS)* are the foremost common vulnerability types in *PyPi*, which are significantly different from the common vulnerability types in ML libraries studied in this paper. Zimmerman et al. [43] conducted an empirical study on *npm* ecosystem to analyze the dependencies among public users of packages, their maintainers, and corresponding public security reports. They find that a single point of failure is the primary vulnerability of *npm* because *npm* packages are often not maintained constantly, which makes large codebases vulnerable. The significant difference with the studied ML libraries is the mitigation strategies where for example, in *npm*, trusted maintainers and code vetting process are two promising fixing strategies.

#### 7.1.3 Vulnerabilities in Android applications

There exist many studies on the vulnerabilities of Android applications [15, 19, 23].

Mazura [23] conducted a large-scale empirical study on Android vulnerabilities by analyzing more than 1k cases from

<sup>9</sup><https://www.npmjs.com/>

<sup>10</sup><https://pypi.org/>

different aspects, including the type of vulnerabilities, the evolution of vulnerabilities, CVSS vendors, the impact of vulnerabilities, and whether they have survived in Android history or not. They find that the significant vulnerability in Android applications is Permissions, Privileges, and Access Controls. Linares et al. [19] characterized different types of vulnerability that may affect android apps, the affected sub-systems, and the time it takes to fix vulnerabilities. Similar to [23], they also mined vulnerabilities from the Android Security Bulletins and the CVE portal. They find that *Memory* and *Data* are the significant types of vulnerability in Android applications. Jimenez et al. [15] performed an empirical study to analyze vulnerabilities of Android applications reported in the National Vulnerability Database. They found that *Missing/incorrect implementation of features* is the dominating vulnerability type. Different from the above studies, we explore vulnerabilities in ML libraries in this paper.

## 7.2 Studies on ML Bugs

### 7.2.1 Studies on ML API Usage Bugs

Islam et al. [13] 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 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. [39] 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. [12] provided an extensive and comprehensive taxonomy of faults in DL libraries. They focused on TensorFlow, Keras, and PyTorch for their study. The notable difference of 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 with our work. First, our study merely focuses on Github commits while their study also mined data from Stackoverflow posts. Second, they analyzed general bugs of DL libraries while we studied security vulnerabilities reported in CWE and CVE portals.

### 7.2.2 Studies on ML Implementation Bugs

Thung et al. [35] studied data from three popular java-based ML libraries to characterize bugs related to the implementation of such tools. Such data are linked to bug reports and bug repositories of the subject programs extracted from the JIRA issue tracking system. Consequently, they came up with 500 bugs and addressed the research questions. They find that algorithmic relayed bugs are the most prevalent in the studied ML libraries. Jia et al. [14] 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. The significant finding of their work is that root causes and symptoms of bugs in TensorFlow are similar to traditional software systems. The most related papers to our study are the studies conducted by Franco et al. [8] and Shen et al. [32]. Franco et al. [8] 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. Also, our analysis is more comprehensive because, besides **Numeric**, we introduce multiple significant vulnerabilities that are common in numerical and ML libraries. Shen et al. [32] proposed a comprehensive study on DL compiler bugs by manually analyzing 596 bugs from TVM from Apache, Glow from Facebook, and nGraph from Intel. They find that type-related bugs are the foremost common bugs in DL compilers. Despite these efforts, the characteristics of software security vulnerabilities have not been well studied, which is the main contribution of this work.

## 8 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 is to help developers of such libraries design and develop vulnerability detection and debugging techniques to increase their quality and reliability. To achieve this goal, we manually analyzed 596 commits from five widely used ML libraries, including TensorFlow, PyTorch, Scikit-Learn, Pandas, and Numpy. The outcome of this study is 19 vulnerability types, 18 root causes, 5 symptoms, 22 fixing patterns, 4 fixing scales, and ultimately 8 findings. Based on these findings, we further provide a set of actionable guidelines to developers and the community to design and develop software vulnerability detection and debugging techniques to increase ML libraries' security.

### Availability

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

### References

- [1] Keras. <https://github.com/fchollet/keras>, 2015. GitHub.

- [2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.
- [3] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 446–457. IEEE, 2021.
- [4] Görkem Algan and Ilkay Ulusoy. Image classification with deep learning in the presence of noisy labels: A survey. *Knowledge-Based Systems*, 215:106771, 2021.
- [5] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2018.
- [6] Ram G Athreya, Srividya K Bansal, Axel-Cyrille Ngonga Ngomo, and Ricardo Usbeck. Template-based question answering using recursive neural networks. In *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, pages 195–198. IEEE, 2021.
- [7] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek 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*, pages 257–268, 2014.
- [8] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. A comprehensive study of real-world numerical bug characteristics. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 509–519. IEEE, 2017.
- [9] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almann, Yuan Xia, and Qi Alfred Chen. A comprehensive study of autonomous vehicle bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 385–396, 2020.
- [10] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.
- [11] Joo-Wha Hong, Yunwen Wang, and Paulina 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*, 36(18):1768–1774, 2020.
- [12] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1110–1121, 2020.
- [13] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh 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*, pages 510–520, 2019.
- [14] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software*, 177:110935, 2021.
- [15] Matthieu Jimenez, Mike Papadakis, Tegawendé F Bisseyandé, and Jacques Klein. Profiling android vulnerabilities. In *2016 IEEE International conference on software quality, reliability and security (QRS)*, pages 222–229. IEEE, 2016.
- [16] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112. IEEE, 2016.
- [17] Raturaj Kulkarni, Shruti Dhavalikar, and Sonal Bangar. Traffic light detection and recognition for self driving cars using deep learning. In *2018 Fourth International Conference on Computing Communication Control and Automation (ICCCUBEA)*, pages 1–4. IEEE, 2018.
- [18] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2201–2215, 2017.
- [19] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 2–13. IEEE, 2017.
- [20] Yunqiu Lv, Bowen Liu, Jing Zhang, Yuchao Dai, Aixuan Li, and Tong Zhang. Semi-supervised active salient object detection. *Pattern Recognition*, 123:108364, 2022.
- [21] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

- [22] Farzaneh Mahdisoltani, Guillaume Berger, Waseem Gharbieh, David Fleet, and Roland Memisevic. Fine-grained video classification and captioning. *arXiv preprint arXiv:1804.09235*, 5(6), 2018.
- [23] Alejandro Mazuera-Rozo, Jairo Bautista-Mora, Mario Linares-Vásquez, Sandra Rueda, and Gabriele Bavota. The android os stack and its vulnerabilities: an empirical study. *Empirical Software Engineering*, 24(4):2056–2101, 2019.
- [24] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.
- [25] Shervin Minaee and Zhu Liu. Automatic question-answering using a deep similarity neural network. In *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 923–927. IEEE, 2017.
- [26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.
- [27] Ripon Patgiri. A taxonomy on big data: Survey. *arXiv preprint arXiv:1808.08474*, 2018.
- [28] 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*, 12:2825–2830, 2011.
- [29] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 383–387. IEEE, 2019.
- [30] Sebastian Ramos, Stefan Gehrig, Peter Pinggera, Uwe Franke, and Carsten Rother. Detecting unexpected obstacles for self-driving cars: Fusing deep learning and geometric modeling. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1025–1032. IEEE, 2017.
- [31] Pradeep Kumar Roy. Deep neural network to predict answer votes on community question answering sites. *Neural Processing Letters*, 53(2):1633–1646, 2021.
- [32] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang 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*, pages 968–980, 2021.
- [33] Ramesh Simhambhatla, Kevin Okiah, Shravan Kuchkula, and Robert Slater. Self-driving cars: Evaluation of deep learning techniques for object detection in different driving conditions. *SMU Data Science Review*, 2(1):23, 2019.
- [34] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical software engineering*, 19(6):1665–1705, 2014.
- [35] Ferdian Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 271–280. IEEE, 2012.
- [36] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *2012 34th international conference on software engineering (ICSE)*, pages 386–396. IEEE, 2012.
- [37] Song Wang and Nachiappan Nagappan. Characterizing and understanding software developer networks in security development. In *The 32nd International Symposium on Software Reliability Engineering (ISSRE 2021)*, 2021.
- [38] Dumidu Wijayasekara, Milos Manic, Jason L Wright, and Miles McQueen. Mining bug databases for unidentified software vulnerabilities. In *2012 5th International conference on human system interactions*, pages 89–96. IEEE, 2012.
- [39] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.
- [40] Zejun Zhang, Yanming Yang, Xin Xia, David Lo, Xiaoxue Ren, and John Grundy. Unveiling the mystery of api evolution in deep learning frameworks a case study of tensorflow 2. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 238–247. IEEE, 2021.

- [41] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. How do python framework apis evolve? an exploratory study. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 81–92. IEEE, 2020.
- [42] Yaqin Zhou and Asankhaya 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*, pages 914–919, 2017.
- [43] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 995–1010, 2019.