Automatic Static Vulnerability Detection for Machine Learning Libraries: Are We There Yet?

Nima Shiri harzevili*, Jiho Shin*, Junjie Wang†, Song Wang*, Nachiappan Nagappan‡
*York University; †Institute of Software, Chinese Academy of Sciences; ‡META
{nshiri,jihoshin,wangsong}@yorku.ca; junjie@iscas.ac.cn; nachiappan.nagappan@gmail.com

***Abstract*—Automatic detection of software security vulnerabilities is critical in software quality assurance. Many static analysis tools that can help detect security vulnerabilities have been proposed. While these static analysis tools are mainly evaluated on general software projects call into question their practical effectiveness and usefulness for Machine Learning (ML) libraries. In this paper, we address this question by analyzing five popular and widely used static analysis tools, i.e., Flawfinder, RATS, Cppcheck, Facebook Infer, and Clang static analyzer, on a curated dataset of software security vulnerabilities gathered from four popular ML libraries, including Mlpack, MXNet, PyTorch, and TensorFlow, with a total of 410 known vulnerabilities. Our research categorizes these tools' capabilities to understand better the strengths and weaknesses of the tools for detecting software security vulnerabilities in ML libraries. Overall, our study shows that static analysis tools find a negligible amount of all security vulnerabilities accounting for 5/410 unique vulnerabilities (0.01%), Flawfinder and RATS are the most effective static checkers for finding software security vulnerabilities in ML libraries. We further identify and discuss opportunities to make the tools more effective and practical based on our observations.**

***Index Terms*—Software vulnerabilities, static detection, machine learning libraries**

I. Introduction

Programming inevitably involves dealing with vulnerabilities in software, which is an aspect that can be frustrating for many developers since detecting and fixing vulnerabilities is time-consuming [1], [2]. To help developers find software vulnerabilities, many static analysis tools have been developed and are now frequently employed by many industries and open-source projects [3]–[6]. Error Prone from Google [7], Infer from Facebook [8], and SpotBugs [9], the successor to the widely used FindBugs tool [10], are examples of popular static analysis tools. These tools are usually developed as an analytical framework based on static analysis and are capable of scaling to large applications.

Previous research has examined static analysis tools on traditional projects from different aspects [11]–[16]. The major limitation of previous studies is that the datasets used for the empirical evaluation of static analysis tools are not real-world examples, they are not able to replicate new and sophisticated security vulnerabilities patterns. Recently, Lipp et al. [16] addressed the limitations by proposing an empirical evaluation of static analysis tools on real-world datasets collected from CVE records gathered from 27 projects containing 1.15 million lines of code. Their results showed that state-of-the-art tools can detect in-between 20% and 53% of the vulnerabilities in a benchmark set of real-world programs. However, it has not been determined if the findings of these studies on conventional projects are applicable to ML libraries. Finding real-world security vulnerabilities in ML libraries is critical for a couple of reasons. First, ML libraries have been widely used in many fields in the past decades, such as image classification [17], [18], big data analysis [19], pattern recognition [20], autonomous driving [21]–[23], and natural language processing [24]–[26]. Failure to detect vulnerabilities in these ML libraries might have devastating implications, such as traffic accidents [27]. Second, gaining an understanding of the benefits and drawbacks of the static analysis tools that are currently in use will direct future work toward the design of new methodologies or tools for detecting ML-specific vulnerabilities.

In this paper, we set out to investigate the effectiveness of five popular static analysis tools on ML libraries. Specifically, we conduct an empirical analysis utilizing 410 real-world security vulnerabilities collected from four widely-used popular ML libraries to address the issue of how many vulnerabilities from these ML libraries can be detected by static analysis tools and why they miss detecting real-world security vulnerabilities of ML libraries. The granularity of our collected dataset is at the commit level in which each commit may contain multiple files involved in fixing the vulnerability. We select five popular and open-source static analysis tools as the research subjects, i.e., Flawfinder [3], [16], [28]–[34], RATS [3], [33], [35], Cppcheck [3], [29], [34], [36], Facebook Infer [8], [37], [38], [38], and Clang static analyzer [3], [39]–[42]. We run these tools on the modified files from the collected commits in our dataset. For each of these modified files, we extract two versions, i.e., the source code before the security vulnerability was introduced and the source code after the vulnerability was fixed. Next, we run the static analysis tools on the source code before the vulnerability was introduced to identify any warnings or issues present in that version. We follow the diff-based mapping technique used in previous research [43] to identify the warnings that fall within the code changes made during the vulnerability-fixing commit. After that, we proceed to perform a manual inspection of the identified vulnerability candidates. The purpose of this manual inspection is to avoid any potential coincidental matches or false positives. By carefully reviewing the vulnerability candidates, we ensure the accuracy and reliability of our vulnerability detection process.

In addition, we have also investigated static analysis tools given a set of program analysis criteria such as input representation, pattern matching techniques, and source code element sensitivity. Specifically, we manually examined the documentation as well as the source code of these static analysis tools to understand how they function. We also study the characteristics of security vulnerabilities in ML libraries and understand their patterns. As a result, we provide future directions to improve existing static checkers to deal with specific security vulnerabilities in ML libraries.

This paper has the following contributions:

- We present the first empirical study on the investigation of current static analysis tools on four widely-used machine learning libraries.
- We identify gaps between existing static analysis tools techniques and software security practices on machine learning libraries.
- We explore the capabilities and limitations of each static analysis tool, as well as suggest future ideas for enhancing the effectiveness of these tools in real-world settings to detect software security vulnerabilities specific to ML libraries.
- We release the dataset of our experiments to help other researchers replicate and extend our study[1].

II. Background & Motivation

ML libraries and traditional software differ in various aspects such as complexity, data dependency, and performance characteristics. Overall, ML libraries tend to be more complex than traditional software as ML often involves complex mathematical models and algorithms [44]–[47]. In addition, as ML models are heavily dependent on the quality and quantity of the data used in their training process, most ML libraries provide tools for data pre-processing, cleaning, and normalization that are not available in traditional software. ML libraries are designed to take advantage of modern hardware, including GPUs and specialized processors, to optimize performance. Traditional software, on the other hand, is generally optimized for general-purpose CPUs.

These differences introduce different vulnerability patterns compared to traditional software. Figure 1a and Figure 1b show two examples of buffer overflow in TensorFlow library[2] and Linux kernel[3] respectively. The root cause of buffer overflow in Figure 1a is lack of proper input validation when getting a non-scalar resource tensor while the root cause of buffer overflow in Figure 1b is using *snprintf()* instead of using *scnprintf()* as *snprintf()* is not a secure built-in C API. We can see that there are two instances of the same vulnerability (i.e., buffer overflow) that have distinct root causes. To detect a heap buffer overflow example in the TensorFlow library as shown in Figure 1a, existing static analysis tools must follow two crucial steps. Firstly, these tools need to recognize the TensorFlow-specific macro checkers (*CHECK_EQ*) by incorporating the macro call signature into their internal database of risky APIs. Secondly, they must conduct data flow analysis to trace the flow of data from client APIs to the backend implementation. In this example, the vulnerable parameter is *ndims* which has been coming from the client API usage. Unfortunately, we discovered that the current tools lack the capability to perform these essential steps, rendering them inadequate for detecting ML-specific vulnerabilities.

The different symptoms and root causes of the same type of security vulnerabilities from ML libraries and traditional software motivate us to look deeper into understanding the performance of current static analysis tools on ML libraries.

III. Study design

In this section, we first discuss the criteria for ML library selection(Section III-A). Then we elaborate on how we curated a dataset of real-world vulnerabilities of ML libraries (Section III-B). Afterward, we introduce the static analysis tools used in this paper (Section III-C), finally, we describe the procedure of applying the static analysis tools to our curated real-world data (Section III-D).

A. ML Library Selection

Our study is on the basis of four widely used ML libraries selected based on a set of inclusion criteria: 1) libraries should be open source and available to the public, 2) they should be under active development, 3) libraries should have the implementation of classical ML as well as state-of-the-art DL models, 4) libraries should support different tasks in the common ML workflow. The outcome of applying these criteria is the following ML libraries; TensorFlow [48], PyTorch [49], MXNet [50], and Mlpack [51]. Our filters also exclude some well-known libraries including Caffe, Theano, and Keras due to the fact that the data extraction mechanism utilized in this work did not retrieve enough real-world security vulnerabilities from their repositories due to a lack of sufficient vulnerability records in the GitHub repository of these libraries.

B. Collection of Vulnerabilities from ML Libraries

For each of the studied ML libraries, we followed the approach proposed by Zhou et al [52] to extract vulnerability-related commits. Note that we extract all commits in the default branch of each library since the starting date of the development. Specifically, we use their regular expression rules, including expressions and keywords related to security issues, to collect security vulnerability-fixing commits. As a result, we collected around 5k commits. Note that, the collected commits might contain noises due to the fact that there may be coincidental matches between vulnerability keywords and the keywords inside the commit message and title [52]. To remove noises, we further conducted a manual inspection on each commit. In our manual inspection, two authors began evaluating extracted commits simultaneously. The authors analyzed the title, message, merged pull requests,

[1] https://anonymous.4open.science/r/ISSRE2023SATS-E255/README.md

[2] https://github.com/tensorflow/tensorflow/commit/13ef0af4867477cdda7e0b294e61560c2952df42

[3] https://github.com/torvalds/linux/commit/5549af7f42916c0d7e78a0e423ac667e27eaac3e

```
gtl::InlinedVector<npy_intp, 4> dims(ndims);
if (TF_TensorType(tensor) == TF_RESOURCE) {
  dims[0] = TF_TensorByteSize(tensor);
  CHECK_EQ(ndims, 0)}
     << "Fetching of non-scalar
     resource tensors is
     not supported.";
  dims.push_back(TF_TensorByteSize(tensor));
  *nelems = dims[0];
} else {
  *nelems = 1;
```

(a) An example of buffer overflow in TensorFlow library.

```
chip = get_chip_info(sdev->pdata);
for (i=0; i<HDA_EXT_ROM_STATUS_SIZE; i++){
value = snd_sof_dsp_read(sdev, HDA_DSP_BAR,
 chip->rom_status_reg + i * 0x4);
  len += snprintf(msg+len, sizeof(msg)-len,
 " 0x%x", value);
  len += scnprintf(msg+len, sizeof(msg)-len,
 " 0x%x", value);
 }
dev_printk(level, sdev->dev,
"extended rom status: %s", msg);
```

(b) An example of buffer overflow in Linux kernel.

Fig. 1: This figure shows two examples of buffer overflow from the TensorFlow library and Linux kernel.

TABLE I: Characteristics of bug fixing commits (BFCs) used in this paper.

Library	Language	LOC(∼)	# BFCs
Mlpack	C++	340K	47
MXNet	C++/Python	362K	60
PyTorch	C++/Python	3.8M	58
TensorFlow	C++/Python	567K	245
Overall		5M	410

and linked issues[4] of each vulnerability fixing commit and remove the commits that are not related to a registered security vulnerability in CWE website[5]. As a result, we collected 410 vulnerability-fixing commits across the four projects that were investigated(shown in Table I).

In this paper, we use the term *vulnerability* in a general sense to refer to any kind of software defect, including security vulnerabilities, logical vulnerabilities, and performance vulnerabilities.

C. Running Static Analysis Tools

In this paper, we select five widely used and open source static analysis tools including Facebook Infer [8], Clang static analyzer [39], Cppcheck[6], Flawfinder (Flawfinder [53], and RATS[7]). The selection of five tools used in our study is based on a comprehensive review of recently published papers on the empirical evaluation of static analysis tools [14], [16]. Please note that three of the tools are under active development including Facebook Infer [8], Clang static analyzer [39], and Cppcheck[8]. The two remaining tools (Flawfinder [53] and RATS[9]) are not actively developing but they have been frequently used in software vulnerability detection [3], [54], [55].

The static analysis tools used in this work have several common characteristics. For example, Flawfinder and RATS treat the input source code as a text sequence while Cppcheck, Infer, and Clang static analyzers convert the source code into an intermediate representation. All tools use a built-in database of patterns that are used to detect security vulnerabilities. For instance, RATS, Flawfinder, and Cppcheck all use a database of C/C++ system functions that are known to have security vulnerabilities such as buffer overflow or format string issues. while Infer uses a bi-abduction inference and analysis in order to find vulnerable source code statements. Clang static analyzer work by parsing the source code and checking it against a set of predefined checks, or *linters*.

For the capabilities of these five static analysis tools used in this paper, following existing work [15], we take into consideration the following three different types of program analysis features, i.e., input representation, matching strategies (*intraprocedural* and *interprocedural*), and sensitivity to program elements including *flow*, *context*, *field*, *object*, *path*, and *field*. We investigated the tools in relation to the aforementioned analytical program analysis features. During this procedure, we manually evaluated the tools' source code and documentation, which demonstrated several types of behaviors to check the tools' capabilities and limits. In the following paragraphs, each tool is explained in detail. Table II shows the capabilities of static analysis tools investigated.

Flawfinder [28]: is a static analysis tool that scans a program for potential security bugs by using a database of known unsafe C/C++ functions. Flawfinder can detect problems with race conditions and system calls in addition to `printf()` and normal string manipulation operations, based on an internal database that contains C/C++ routines that are known to have security bugs in their design.

RATS [35]: An open-source static analyzer that is able to analyze code bases written in C, C++, Perl, PHP, and Python. Similar to Flawfinder, it uses an internal database of risky C/C++ API signatures and uses a keyword-matching approach to find and mark them as vulnerable in the target source code.

Cppcheck [36]: Cppcheck offers one-of-a-kind code analysis to find defects and focuses on finding undefined behavior as well as risky coding structures. The objective is to generate an extremely low number of false positives.

Infer [8]: is a static analysis tool that is developed by Facebook. It searches for a wide variety of vulnerabilities

[4]Some vulnerability fixing commits fix opened issues. Thus, the authors further analyzed them for noise removal.

[5]https://cwe.mitre.org/

[6]https://cppcheck.sourceforge.io/

[7]https://github.com/andrew-d/rough-auditing-tool-for-security

[8]https://cppcheck.sourceforge.io/

[9]https://github.com/andrew-d/rough-auditing-tool-for-security

TABLE II: Characteristics of tools based on different program analysis factors.

Tool	Version	Input representation			Pattern matching		Sensitivity				
		Text	AST	Other	Intraprocedural	Interprocedural	Context	Field	Object	Data Flow	Control Flow
Flawfinder	2.0.19	✓	-	-	×	×	×	×	×	×	×
RATS	2.4	✓	-	-	×	×	×	×	×	×	×
Cppcheck	2.7	-	✓	-	✓	×	✓	×	×	*	*
Infer	1.1.0	-	-	✓	✓	✓	✓	✓	N.A	✓	✓
Clang static analyzer	14.0.0	-	✓	-	✓	✓	✓	✓	✓	✓	✓

in programs written in Java, C/C++, and Objective-C. Bi-abduction analysis is one of the methods that Infer employs in order to locate vulnerabilities such as deadlocks, memory leaks, and null pointer dereference. In order to perform analysis, Infer requires a set of compilation commands for each file. Infer is an interprocedural analysis tool which means that it allows keeping track of objects and variables between methods, and also global variables.

Clang static analyzer [39]: Clang static analyzer is built on top of the LLVM project, which offers a set of modular and reusable compiler and toolchain technologies. It is an extensible framework for C/C++ code linting that may be used to enforce coding standards, conduct static analysis, and discover probable errors. It operates by scanning the source code and comparing it to a collection of predefined checks, known as linters. The checks are built as separate modules, making it simple to add new checks or modify current ones.

Note that the static analysis tools used in this paper have capabilities to report non-vulnerable related warnings, e.g., styling or refactoring issues, as well as reporting general vulnerabilities which are not related to software vulnerabilities that have a unique id in CWE website[10]. In order to reduce false positive rates and prevent static checkers to produce related warnings to vulnerabilities, we have configured them to merely report warnings that are strongly related to software security vulnerabilities. Configuring the tools to report all types of vulnerabilities can introduce a large number of false positives (non-security-related code), which require much more time for manual verification. Please note that RATS does not generate styling issues. In addition, we manually studied Flawfinder's built-in database and discovered that all rules have a unique id associated with CWE records. As a result, we solely used this setup for Infer, Cppcheck, and Clang static analyzer.

D. Identification of Vulnerable Candidates

One of the main challenges in the identification of vulnerable candidates is to check whether the reported warnings by static analysis tools are corresponding with the issued vulnerability class or not. To identify vulnerable candidates, we use the following steps. First, we performed automatic filtering based on the commits' diff information as used in existing studies [14], [15]. Afterward, we further analyzed the reported warnings that involve the commit using manual inspection.

[10]https://cwe.mitre.org/

1) Automatic filtering: We filter reported warnings by using a diff-based mapping technique which uses code diffs between vulnerable and fixed programs in a fixing commit [43]. First, it computes a set of lines in the vulnerable program that is flagged with at least one warning. Then, it checks whether the flagged line numbers overlap with the changed lines in the fixing commit. If the flagged lines overlap with the code change, the warning is considered a vulnerability candidate. Otherwise, it is not a candidate for manual inspection. For example, Figure 2 shows an example of a warning generated by Flawfinder that is considered to be a vulnerable candidate since the line number reported in the warning overlaps with the line number in the code change of the vulnerability fixing commit.

2) Manual verification: In this step, we manually scan all candidates and check the warning messages against the vulnerable and clean versions of the code to eliminate potential accidental matches. The first two authors examine the warnings and vulnerability-fixing commits simultaneously. They manually analyze the warning message, the CWE-ID, and the line number where the vulnerability is reported. Based on the above information, they review the code changes to confirm whether the warning is related to the vulnerability being addressed in the commit. Note that, in the cases where the static analysis tools reported a warning at a line outside those changed lines in the vulnerability-fixing commit, we manually checked the warning. In addition, we have also performed a backward analysis in which we traced all variables and function calls within the code changes up to the marked line that was outside the code changes. If the marked line was associated with the code changes, we considered the corresponding warning as a potential vulnerability. If there is any disagreement, they flag the commit and the corresponding warning for further manual verification in the next round. They repeat this process several times until all warnings and commits have been reviewed.

During our manual inspection, we also audit the following information: 1) True Positive (TP): the vulnerability described in the warning precisely matches with the vulnerability reported for the commit; False Negative (FN): a potential warning mistakenly predicted to be a false alarm. In this paper, we calculate True Positive Rate (TPR) and False Negative Rate (FNR) as $TPR = \frac{TP}{TP+FN}$ and $FNR = \frac{FN}{FN+TP}$ respectively.

IV. Experimental Results

In this section, we present and discuss our analysis results to address the following three research questions.

RQ1 (Detection Capability): How many warnings are reported by the studied static analysis tools?

```
43   for (int i = 0; i < client->device_count();
44   se::StreamExecutorConfig config;
45   config.ordinal = i; config.device_options.no
46   ["host_thread_stack_size_in_bytes"] =
47    -absl::StrCat(2048  1024);
48    +absl::StrCat(8192  1024);
49   TF_ASSIGN_OR_RETURN(se::StreamExecutor * exe
50   platform->GetExecutor(config));
```

(a) bug fixing commit.

```
Examining /cpu_device.cc
FINAL RESULTS:
cpu_device.cc:47:
[4] (buffer) StrCat:
Does not check for buffer overflows when
concatenating to destination
[MS-banned] (CWE-120).
absl::StrCat(2048 * 1024)
```

(b) reported warning by Flawfinder.

Fig. 2: An example of vulnerability fixing commit in TensorFlow library in which the developer has increased the thread stack size from 2048 to 8192 to prevent buffer overflow vulnerability. Figure 2b shows the generated warning by Flawfinder in which there is a possible buffer overflow at line 47.

TABLE III: Distribution of reported warnings.

Library	Flawfinder	RATS	Cppcheck	Infer	Clang static analyzer
Mlpack	3	22	1	0	0
MXNet	10	6	7	0	0
PyTorch	5	1	2	4	16
TensorFlow	50	139	0	2	254
Overall	68	168	10	6	270

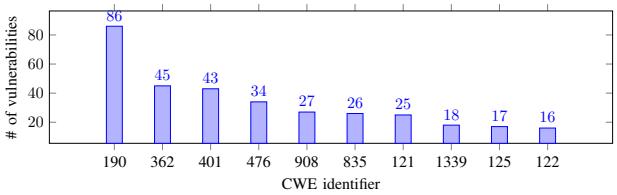

Fig. 3: The top 10 vulnerability types in our dataset.

RQ2 (Detection Effectiveness): How effective are static analysis tools at detecting real-world vulnerabilities in ML libraries?

RQ3 (Root Cause): What are the root causes of missing real-world vulnerabilities in ML libraries?

For answering these RQs, we execute Flawfinder, RATS, Cppcheck, Infer, and Clang static analyzer on our dataset of 410 real-world bugs across the four studied ML libraries. The output of bug detectors, i.e., generated warnings, are automatically parsed to extract relevant information including bug types, line numbers, and warning messages. All data and scripts for conducting the experiments described in this section are accessible to the public.

A. RQ1: Detection Capability

Experiment setup. To answer this question, we directly run these five static analysis tools on the four ML libraries, we record all the reported vulnerabilities (including the detailed location and CWE information) by these tools for further analysis. For each of the reported vulnerabilities, we further manually check whether it's a true vulnerability or a false positive. Please note that the granularity of our collected dataset is at the commit level in which each commit may contain multiple files involved in fixing a vulnerability. For each commit, we extract modified files, i.e., affected by the vulnerability. For each modified file, we extracted two versions, i.e., source code before the vulnerability and source code after fixing the vulnerability. We then run the tools on the source code before the vulnerability and generate the warnings.

Table III displays the number of vulnerability types extracted from warnings reported by each static analysis tool across the four ML libraries. Specifically, among the four ML libraries, TensorFlow has the highest number of reported warnings, i.e., in total 445 warnings. *Mlpack* has the lowest number of reported warnings overall, with only 26 warnings across all tools. Among the static analysis tools, *Clang static analyzer* reports the highest number of warnings for each library. *Infer* only reports six warnings.

We further illustrate the top 10 vulnerability types and their distribution in our curated dataset in Figure 3. The chart shows that the most common vulnerability type is CWE-190 (Integer Overflow or Wraparound), with 86 occurrences in the studied ML libraries. The next most common types are CWE-362 (Race Condition), CWE-401 (Missing Release of Resource after Effective Lifetime), and CWE-476 (NULL Pointer Dereference), with 45, 43, and 34 occurrences, respectively. The least common vulnerability type in the chart is CWE-122 (Heap-based Buffer Overflow), with only 16 occurrences.

Table IV also shows the number of false positive warnings out of all warnings generated given each true vulnerability type. Specifically, Flawfinder reported a total of 50 warnings for CWE-120, but only 3 of them were accurate warnings. In addition, it reported a single warning for CWE-190 that was indeed accurate. For CWE-20, Flawfinder reported 5 true positive warnings. Regarding CWE-190, Flawfinder reports one warning which is true positive. RATS produced 27 warnings for CWE-121, but only 1 of them is a true warning. For CWE-122, RATS flagged 8 warnings, but merely 3 of them are true positive. All warnings generated by Cppcheck and Infer are false positives suggesting that they are not effective at detecting real-world security vulnerabilities in the studied ML libraries.

TABLE IV: Distribution of vulnerability types reported by the four static vulnerability detectors. The warnings generated by Clang static analyzer do not have a CWE number, so we exclude it in this table. Numbers in the brackets are false positive warnings. The numbers outside of the brackets are the total number of generated warnings.

CWE-ID	# True Bugs	Flawfinder	RATS	Cppcheck	Infer	Overall
CWE-120	-	50 (47)	0	0	0	50
CWE-121	25	0	27 (26)	0	0	27
CWE-122	24	0	8 (5)	0	1 (1)	9
CWE-362	45	7 (7)	2 (2)	0	0	9
CWE-327	-	7 (7)	0	0	0	7
CWE-367	-	7 (7)	0	0	0	7
CWE-20	7	5 (0)	0	0	0	5
CWE-476	34	0	0	5 (5)	0	5
CWE-807	-	4 (4)	0	0	0	4
CWE-126	-	3 (3)	0	0	0	3
CWE-330	-	0	3 (3)	0	0	3
CWE-457	1	0	0	0	2 (2)	2
CWE-563	-	0	0	0	1 (1)	1
CWE-667	-	0	0	1 (1)	0	1
CWE-758	-	0	0	1 (1)	0	1
CWE-190	86	1 (0)	0	0	0	1
CWE-398	-	0	0	1 (1)	0	1
CWE-119	-	1 (1)	0	0	0	1
CWE-833	15	0	0	0	1 (1)	1
CWE-401	43	0	0	0	0	0
CWE-908	27	0	0	0	0	0
CWE-835	26	0	0	0	0	0
CWE-1331	18	0	0	0	0	0
CWE-125	17	0	0	0	0	0
CWE-369	10	0	0	0	0	0
CWE-703	4	0	0	0	0	0
CWE-191	3	0	0	0	0	0
CWE-705	3	0	0	0	0	0
CWE-415	3	0	0	0	0	0
CWE-704	2	0	0	0	0	0
CWE-416	2	0	0	0	0	0
CWE-840	2	0	0	0	0	0
CWE-787	2	0	0	0	0	0
CWE-439	2	0	0	0	0	0
CWE-628	1	0	0	0	0	0
CWE-241	1	0	0	0	0	0
CWE-255	1	0	0	0	0	0
CWE-197	1	0	0	0	0	0
CWE-252	1	0	0	0	0	0
CWE-706	1	0	0	0	0	0
CWE-1006	1	0	0	0	0	0
CWE-561	1	0	0	0	0	0
CWE-475	1	0	0	0	0	0
Overall	410	85	40	8	5	138

Finding 1: The number of warnings produced by the five static analysis tools varies significantly and is mainly determined by their detection mechanisms. While these tools can also generate false positives, human review is still necessary to determine whether reported warnings are real-world vulnerabilities that require attention.

B. RQ2: Detection Effectiveness

Experiment setup. To address this question, the first two authors of this paper manually reviewed the generated warnings (vulnerability candidates identified using the automatic filtering approach explained in subsection III-D1) per file for each vulnerability-fixing commit. The team conducted a thorough analysis of the warning message, the CWE-ID, and the specific line numbers that can indicate where the vulnerability was identified. Using this information, they further examined the code modifications to verify that the warning is indeed related to the vulnerability being addressed in the particular update. If any discrepancies arise, they highlighted the corresponding warning and committed to additional examination in the subsequent review cycle. This process is repeated until all the warnings and code changes have been assessed.

TABLE V: Performance of the studied static vulnerability detectors on the four ML libraries.

Tool	TPR	FNR
Flawfinder	0.04	0.95
RATS	0.03	0.97
Cppcheck	0	1
Infer	0	1
Clang static analyzer	0	1

TABLE VI: Vulnerabilities detected by Flawfinder and RATS.

Tool	Actual bug	Reported bug(# warnings)	Library	Filename
Flawfinder	CWE-125	CWE-20(4)	PyTorch	*simd.h*
Flawfinder	CWE-197	CWE-120(2)	PyTorch	*decode_padded_raw_op.cc*
Flawfinder	CWE-122	CWE-20(1)	MXNet	*image-classification-predict.cc*
Flawfinder	CWE-190	CWE-190(1)	MXNet	*image_iter_common.h*
Flawfinder	CWE-121	CWE-120(1)	TensorFlow	*cpu_device.cc*
RATS	CWE-122	CWE-121(1)	MXNet	*image-classification-predict.cc*
RATS	CWE-190	CWE-122(1)	PyTorch	*THTensor.cpp*
RATS	CWE-197	CWE-122(2)	PyTorch	*decode_padded_raw_op.cc*

Results. Table V shows the performance of each static analysis tool regarding True Positive Rate (TPR) and False Negative Rate (FNR) values. As we can see from the table, Flawfinder has a TPR of 0.049, which means it correctly identifies only 4.95% of the actual vulnerabilities, and an FNR of 0.95, indicating that it fails to detect 95.05% of actual vulnerabilities. RATS has a TPR of 0.030 meaning that it correctly identifies only 3.07% of the actual vulnerabilities, and a FNR of 0.97, which means it fails to detect 97.01% of the actual vulnerabilities. The rest of the tools have a TPR of 0 which states that they fail to detect any actual vulnerabilities, and an FNR of 1, which means they incorrectly identify all actual vulnerabilities as negatives. Overall, all the experimented tools have relatively low performance in terms of identifying actual vulnerabilities in ML libraries, as they all have very low TPR values or fail to detect any real-world vulnerabilities at all.

Table VI shows the detailed characteristics of vulnerabilities detected by Flawfinder and RATS. We can see that Flawfinder and RATS can identify vulnerabilities in different code repositories, including MXNet, TensorFlow, and PyTorch. The vulnerabilities are categorized according to their types (i.e., CWE numbers). For example, Flawfinder detected a CWE-122 and a CWE-190 in MXNet. RATS also detected a CWE-122 in MXNet, as well as a CWE-190 in PyTorch.

```
auto input_shape = c->input(0);
auto input_h_shape = c->input(1);
auto seq_length = c->Dim(input_shape, 0);
// assumes rank >= 2
auto batch_size = c->Dim(input_shape, 1);
// assumes rank >= 3
auto num_units = c->Dim(input_h_shape, 2);
```

Fig. 4: An example of heap buffer overflow in TensorFlow library.

Finding 2: Overall, the effectiveness of the tools is quite poor in discovering real-world ML software vulnerabilities. Flawfinder and RATS, which are the most effective static checker, discovered 4 unique vulnerabilities out of a total of 410 vulnerabilities in our dataset.

C. RQ3: Root Cause of Missing Real-world Vulnerabilities

To address this research question, we manually analyzed and reviewed the documentation of tools as well as their code base. Then, we organized the reasons why these tools miss detect so many vulnerabilities.

1) Flawfinder and RATS: Since Flawfinder and RATS manifest very similarly in vulnerability detection, Flawfinder's specific reasons also apply to RATS.

Issues with Soundness Strategy. Flawfinder and RATS have a significant limitation in their approach to vulnerability detection, known as the soundness strategy. The soundness strategy of Flawfinder and RATS is that they use pre-defined risky C/C++ APIs in their internal databases to search the source code and label every matched API call (already registered in the database) as a potential vulnerability. Although ML libraries often make extensive use of C/C++ APIs in their backend implementation, the vulnerabilities in these libraries do not primarily originate from these APIs. Instead, the vulnerabilities in ML libraries tend to exhibit more complex patterns (require a deeper source code analysis in order to be detected), which are not effectively captured by the simplistic approach of flagging all API calls as vulnerable [44], [45], [45]–[47], [56]–[58].

Lack of ML-specific Vulnerable API Information. Even though Flawfinder and RATS support more than 200 dangerous C/C++ APIs, they are still incapable of supporting ML-specific vulnerable APIs. One of the reasons they miss detecting real-world security vulnerabilities in ML libraries is that they do not model library-specific API information [44], [47]. For example, to detect *Memory Leak (CWE-401)* in the Mlpack library, the static detectors need to model `CleanMemory()` or `delete_mat` which are Mlpack-specific APIs used for cleaning allocated memories.

Lack of Data Flow and Control Flow Support. Flawfinder and RATS face challenges in detecting numerous vulnerability patterns specific to machine learning (ML) because they lack the capability to model control flow and data flow. Instead, these tools rely on a simplistic keyword-matching approach to identify potentially dangerous C/C++ API calls and flag them as vulnerabilities. This limitation hinders their effectiveness in capturing the intricacies of ML-specific vulnerabilities [44], [47] that involve complex control and data dependencies. For example, lack of validation is a complex root cause pattern that is the major root cause of data type vulnerabilities in ML libraries [44]. In this particular vulnerability pattern, vulnerabilities can arise when utilizing client APIs of ML libraries. If the backend implementation fails to validate or properly handle malicious inputs or malformed values received through these APIs, vulnerabilities can occur. It is crucial to perform appropriate validation and handling of such inputs to mitigate potential issues and ensure the robustness and security of the ML library. For example, Figure 4 is an example of a lack of validation that causes a heap buffer overflow. In this vulnerability, the code assumes `input_shape` and `input_h_shape` have a specific rank, while the rank values should be validated to avoid possible invalid memory access. Flawfinder and RATS are not able to detect such vulnerabilities because they are incapable of modeling data flow dependency and keeping track of `input_shape` and `input_h_shape` to check if these variables are validated or not.

2) Cppcheck: **Limited Buffer or Stack Overflow Checking** Existing work [44] has shown that the root cause patterns of buffer overflow or stack overflow in ML libraries are significantly different compared to that of traditional software. For example, in this commit[11] from the TensorFlow library, the root cause of stack overflow is a very big computation graph of functions and edges. In this example, Each instance of the class `std::shared_ptr<Function>` holds a collection of Edge objects, and each Edge object, in turn, holds a `std::shared_ptr<Function>`. Removing a `std::shared_ptr<Function>` can lead to the cascading deletion of other `std::shared_ptr<Function>` instances, potentially resulting in a stack overflow if the graph has a significant depth. However, Cppcheck performs different strategies to detect buffer overflow and stack overflow vulnerabilities which are not strong enough to detect ML-related overflow vulnerabilities. For example, Cppcheck uses array index checkers to detect buffer overflow which identifies array index operations and performs various checks to detect potential buffer overruns associated with array indexing. More specifically, Cppcheck exhaustively searches for array indexing statements in the code snippet, even if the code is not reachable, without any control flow analysis. This technique is incapable of detecting overflow vulnerabilities which is due to the large computation graph mentioned above. The second technique to detect buffer overflow or stack overflow is to search for C/C++ API calls, similar to Flawfinder and RATS explained earlier. In this technique, Cppcheck examines the API scopes and the corresponding arguments, analyzes the buffer or stack sizes associated with the arguments, and detects

[11] https://github.com/tensorflow/tensorflow/commit/932c4c2364884af52609ea8a86c7232a926d958f

```
inline int MatchingDim(const RuntimeShape&
shape1, int index1, const RuntimeShape&
shape2, int index2) {
 TFLITE_DCHECK_EQ(shape1.Dims(index1),
 shape2.Dims(index2));
 return shape1.Dims(index1);
}
```

Fig. 5: An example of hard-to-detect buffer overflow bug in TensorFlow library.

potential buffer overflow or stack overflow vulnerabilities based on specified minimum size requirements.

Limited Control Flow Analysis One major limitation of Cppcheck is limited control flow analysis, while a strong control flow analysis is required to detect very hard-to-detect vulnerabilities In ML libraries, e.g., memory leaks that have complicated vulnerability patterns. For example, this memory leak vulnerability[12] occurs in the TensorFlow library when decoding malformed PNG image. The memory leak occurs when certain errors in the function implementation cause the execution to be abruptly terminated using the `OP_REQUIRES` macro checker. This termination prevents the proper freeing of allocated buffers stored in the `decode` value. To release these allocated buffers, the function should call `png::CommonFreeDecode(&decode)`. However, due to eager termination, the necessary memory-freeing process is not allowed to occur. This vulnerability is very hard to detect by Cppcheck for two main reasons. First, the TensorFlow-specific API `png::CommonFreeDecode(&decode);` is not registered in the internal database of API symbols which is the firsts step toward detecting this leak. Second, the control-flow analysis supported by Cppcheck is very simple and cannot capture the complex flow analysis in this vulnerability pattern.

3) Infer: **Limited Buffer Overflow checkers** Compared to Cppcheck, Flawfinder, and RATS, Infer has a stronger buffer overflow checker. Infer uses symbolic intervals to handle the range of index values and buffer sizes. Typically, interval analysis involves working with intervals represented as [low, high], where low and high are constants indicating the lower bound and upper bound of the target buffer and indexing range. However, this checker is not effective enough to detect ML-related buffer overflow vulnerabilities. For example, Figure 5 shows an example of hard-to-detect buffer overflow[13] in the TensorFlow library where providing a tensor with larger dimensions as the second argument is critical. To address this issue, the developer has modified the `MatchingDim` function to return the minimum size between the two dimensions.

4) Clang static analyzer: **Limited Rules**. Although Clang static analyzer provides a large number of predefined rules (i.e., there are 25 families of rules implemented in the Clang static analyzer database), it does not cover rules of security vulnerabilities that are relevant to ML libraries. For example, the null pointer checker introduced in Clang static analyzer, i.e., *ore.NullDereference (C, C++, ObjC)* is only able to detect simple null pointer violations[14]. While, in a null pointer vulnerability from TensorFlow library[15], the root cause is lack of checking null pointer Tensors as a function argument. Specifically, the vulnerability traces back to the omission of null pointer checks in relation to Tensors utilized as function arguments. Within this instance, the implicated function lacks a TensorFlow-specific mechanism to preempt the erroneous passage of *val* to the *(GetBundleEntryProto(key, entry))*. In the fixing commit, the developers add *CHECK(val != nullptr);* to resolve the issue.

An additional instance of a constrained rule within the Clang static analyzer is the *unix.Malloc (C)* checker. This checker's role is to identify memory leaks within codebases developed in the C programming language. The mechanics of this Clang static analyzer checker, elaborated upon in the official documentation[16], allow it to uncover memory leaks by employing relatively straightforward patterns. Nevertheless, an illustration of a memory leak within the MXNet library[17] serves to underscore the complexity that can underlie memory leak patterns. In this specific case, a memory leak arises due to the usage of

This initial implementation is subsequently replaced by the developer with

```
std::unique\_ptr <DType>
tmp(new DType[Super::WORKLOAD\_COUNT]);
std::unique\_ptr <DType[]>
tmp(new DType[Super::WORKLOAD\_COUNT]);
```

Finding 3: We identified a set of specific reasons that help explain why the five static analysis tools examined in this paper miss real-world security vulnerabilities in ML libraries including ML software-specific reasons, i.e., **Lack of Implementation Support of Detection Rules**, **Issues with Soundness Strategy**, and **Lack of ML-specific Vulnerable API Information**.

V. Lessons Learned

Our study reveals several interesting findings that can serve as applicable guidelines for improving static analysis tool for ML libraries.

A. Implication for Improving Flawfinder and RATS

We find that Flawfinder and RATS flag every C/C++ API as vulnerable regardless of whether they are actually vulnerable or not. This behavior introduces many false alarms IV. In order to reduce the false alarm rate, numerous extensions to these checkers are required.

Extend the input representation. Flawfinder and RATS represent the source code as a sequence of tokens. Unlike

[12] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23585

[13] https://github.com/tensorflow/tensorflow/commit/8ee24e7949a203d234489f9da2c5bf45a7d5157d

[14] https://clang.llvm.org/docs/analyzer/checkers.htmlcore-nulldereference-c-c-objc

[15] https://github.com/tensorflow/tensorflow/commit/ca170f34d9174d6981850855190a398393aa921e

[16] https://clang.llvm.org/docs/analyzer/checkers.htmlunix-malloc-c

[17] https://github.com/apache/mxnet/commit/b1ad1619d8007dc7dfc0f4ba9cc8720b3e0dd32a

natural language, source code encodes structural information [59], which needs to be considered with rich representation techniques like Abstract Syntax Trees(AST), Control Flow Graphs(CFG), or Data Flow Graphs(DFG) [60]. Having a control flow graph is vital in detecting many ML-related security vulnerabilities including buffer overflow and memory leaks. Control flow graphs allow the tool to model every possible execution patch inside vulnerable programs. For example, to detect the stack overflow bug in this commit [18], the checker needs to traverse control graph to find out that the implementation of `DataTypeString` function is vulnerable due to missing `return DataTypeStringInternal(dtype);`.

Extend the pattern matching. The pattern-matching strategy of Flawfinder and RATS follows naive text-based matching. Text-based matching does not allow the detectors to perform intraprocedural analysis, i.e. the analysis inside the boundary of functions or any compilation units. The lack of performing intraprocedural analysis introduces so many false alarms in the case of detecting ML security vulnerabilities. For example, in order to detect *Memory Leaks (CWE-401)* bug listed in Table VI, the detector needs to perform analysis inside `_bsplmat()` and catch the return value which is the root cause of this bug.

B. Implication for Improving Cppcheck

Cppcheck has been shown to be the most effective static checker for detecting real-world security issues. However, it still lacks a plethora of weaknesses. Cppcheck may be extended in two ways, as discussed in the subsections below.

Support ML Library-Specific Constructs. In order to extend Cppcheck, the developers should handle any ML library-specific constructs or macros within the backend implementation. This may involve extending Cppcheck's macro handling capabilities or creating additional checks to handle these constructs effectively.

Extend the control flow graph. The control flow graph analysis in Cppcheck is very limited, with the following assumptions: all source code statements can be accessed, and the state checkers in the if conditions are always either true or false. As a result, a more advanced control flow graph is required to be able to detect sophisticated vulnerabilities in ML libraries.

C. Implication for Improving Infer

Improve Buffer overrun Checker. While Infer's buffer overrun checker can detect typical buffer overflow patterns, it may not cover all conceivable variations and attacker approaches in ML libraries. The tool may miss sophisticated or unique exploitation methods explained in RQ3.

Isolate checkers. To begin with, the rules for identifying integer overflow in Infer are still in an experimental stage and lack sensitivity to function arguments. Additionally, Infer's dependency on buffer overrun rules to detect integer overflow introduces a limitation. These buffer overrun rules themselves tend to generate a significant number of false alarms, which consequently raises the false alarm rate for identifying integer overflow bugs. Given the intricate nature of the bug pattern for integer overflow in ML libraries [44], it becomes crucial for developers to separate the buffer overrun checkers from the scope of integer overflow.

D. Implication for Improving Clang static analyzer

Extend the Checker for Null Pointer Dereference. Clang static analyzer uses its internal checker *core.NullDereference (C, C++, ObjC)* to detect null pointer dereference bugs in C, C++, and Objective C programs. The checker works well if function arguments take pointer parameters. If the code attempts to dereference the pointer without checking if it is null, it marks it as buggy. However, in terms of ML libraries, null pointer dereference has more sophisticated patterns. For example, in this commit from the TensorFlow library[19], the developer has removed `int64 id = ctx->session_state()->GetNewId();` since `session_state()` is vulnerable to have a null value which results in denial of service via a null pointer dereference. The developers should increase the null pointer checker to cover more corner cases in terms of code elements that may have null values.

VI. Threats to validity

As is the case with every empirical research, there are a few factors that call into question the reliability of the inferences we have taken from our data. One of the major limitations is the choice of ML libraries and static analysis tools. In order to protect ourselves against this risk, we have chosen four extensively used ML libraries, each of which focuses on a different facet of ML development. All ML libraries are open-source projects that are always being developed and improved upon. Regarding the selection of static analysis tools, our major focus is on static detectors that are openly available to the public and are in the process of being developed right now. We used widely used and popular detectors which have been cited by many previous studies in the field of software vulnerability detection [3], [54], [55].

Another possible threat to this study is the mapping technique [14] used to find potential vulnerable candidates. The approach we used in the paper is subject to coincidental matches. for example, the assumption is that if the line number produced in warnings overlaps with the modified lines in a vulnerability-fixing commit, the program will automatically recognize the warning as a candidate for the vulnerability. This is not true in practice since the reported warning may not be connected to the actual vulnerability in the commit, or the changed line may be refactoring the code and does not represent the vulnerability being fixed. The ultimate determination of whether a warning relates to a vulnerability is made by two authors involved in manual inspection and is therefore subjective. To mitigate this

[18] https://github.com/tensorflow/tensorflow/commit/698bc996f7190f5cd836d48d29b8c1b3ddcd37c2

[19] https://github.com/tensorflow/tensorflow/commit/9a133d73ae4b4664d22bd1aa6d654fec13c52ee1

risk, both authors reviewed every possibility for an identified vulnerability when there is no clear evidence.

In this paper, we rely on the assumption that a source code before a vulnerability fix is a vulnerable source code, and the source code after the fix is considered as the vulnerability is fixed. One may apply static analysis tools to source code after the fix and find multiple vulnerabilities. As a result, this assumption serves as the foundation for this paper's goal of determining how many real-world security vulnerabilities the tools identify at the moment of committing modifications.

VII. Related work

A. Software Security Vulnerability Detection

There have been numerous studies focused on software vulnerability detection in the literature [3]–[5], [54], [55], [61]. Cao et al. [4] proposed a deep learning-based vulnerability detection model to detect memory-related statements on their manually curated dataset extracted from 11 projects developed in C/C++. Their proposed model can model structural information which allows the detection of semantic vulnerabilities. Their experiments indicate that the proposed model is superior compared to cutting-edge models as well as static analysis tools. Li et al. [61] developed SySeVR, a deep learning-based vulnerability detector in which syntactic and semantic information contained in source codes are merged as a rich input representation and supplied into the model. They think that this unique representation discovers subtle weaknesses in the source code.

B. Studies on Static Detection Techniques

Habib and Pradel [14] investigated the static analyzers including Infer, ErrorProne, and SpotBugs to figure out what percentage of Defect4j's total bugs can be located by using the aforementioned tools. Both the code diff and the bug report mapping approaches are utilized by the authors. Tomassi [62] They carry out a study in which they examine the similarities and differences between ErrorProne and SpotBugs in order to determine the total number of vulnerabilities that are discovered in a sample of 320 BugSWARM artifacts. SpotBugs was only able to locate a single vulnerability, as the author discovered. Rutar et al. [63] analyzed a small suite of programs with a number of different static analyzers, including PMD, FindBugs, JLint, Bandera, and ESC/Java 2. The authors present a taxonomy of vulnerabilities discovered by each tool, demonstrating that none of the tools can be considered to be more comprehensive than the others. Runtime as well as the total number of warnings generated are the primary focuses of this study. Lipp et al. [16] argued that there are two main limitations to the datasets used for testing static analysis tools. Firstly, the datasets do not accurately represent real-world code and cannot identify new and complex vulnerability patterns. Secondly, the datasets do not classify vulnerabilities based on the Common Weakness Enumeration (CWE) mapping. Hence, they attempted to overcome these limitations by testing static analysis tools on real-world datasets that were collected from CVE records of 27 different projects, totaling 1.15 million lines of code.

Static analysis tools often produce a large number of false alarm warnings which makes manual inspection problematic. To remove such false alarms, Several different approaches to the detection of software security vulnerabilities have been taken up by the industry. Bessey et al. [64] share their insights gained from the process of bringing static analysis tools to the market in their paper. Ayewah et al. [10], [65] discussed the lessons learned via applying FindBug on Google's codebase in which numerous engineers involved through thousands of FindBugs generated warnings, and addressed them by either fixing them or filing reports. They find that most issues were highlighted for fixing, but only a few of them were actually causing significant problems in production. Understanding real-world security vulnerabilities is a crucial first step in enhancing vulnerability detection. Several studies have taken different types of vulnerabilities into account, such as those in the Linux kernel [66], vulnerabilities in concurrency [67], and bugs in correctness [68] and performance [69] in JavaScript.

Our work is very different from the studies that have been done before [14]–[16], [62]. In previous studies, the primary focus has been on the automatic static detection of general bugs in traditional software projects. On the other hand, the primary focus of this paper is on the automatic detection of software security vulnerabilities in widely used ML projects. For example, Integer Overflow (CWE-190) is a major vulnerability in ML libraries as found in [44]. A further distinction lies in the fact that the focus of our work is on the more recent and advanced generation of static analysis tools specifically designed to detect security vulnerabilities in projects written in C/C++.

VIII. Conclusion

This paper addresses the critical task of automatic detection of software security vulnerabilities in ML libraries. The study analyzes the effectiveness of five popular and widely used static analysis tools, namely Flawfinder, RATS, Cppcheck, Facebook Infer, and Clang static analyzer, on a curated dataset of software vulnerabilities gathered from four popular ML libraries. The research categorizes these tools' capabilities, highlighting their strengths and weaknesses in detecting software security vulnerabilities. The study reveals that Flawfinder and RATS are the most effective static checkers for finding security vulnerabilities in ML libraries. However, the overall findings show that the tools detect only a negligible amount of vulnerabilities, accounting for 5/410 (0.01%) of known security vulnerabilities. Based on these observations, the paper also identifies and discusses opportunities to make the tools more effective and practical.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] P. Foreman, *Vulnerability management*. CRC Press, 2019.

[2] 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.

[3] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[4] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: Memory-related vulnerability detection based on flow-sensitive graph neural networks," *arXiv preprint arXiv:2203.02660*, 2022.

[5] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: a deep learning-based fine-grained vulnerability detector," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, feb 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374

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

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

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

[10] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," *IEEE software*, vol. 25, no. 5, pp. 22–29, 2008.

[11] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[12] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu *et al.*, "To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 50–59.

[13] ——, "To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools," *Automated Software Engineering*, vol. 22, no. 4, pp. 561–602, 2015.

[14] 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.

[15] 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.

[16] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 544–555.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] J. Chen, C. Zhang, S. Cai, L. Zhang, and L. Ma, "A memory-related vulnerability detection approach based on vulnerability model with petri net," *Journal of Logical and Algebraic Methods in Programming*, vol. 132, p. 100859, 2023.

[30] J. D. Pereira and M. Vieira, "On the use of open-source c/c++ static analysis tools in large projects," in *2020 16th European Dependable Computing Conference (EDCC)*. IEEE, 2020, pp. 97–102.

[31] C. Mitropoulos, "Employing different program analysis methods to study bug evolution," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1202–1204.

[32] M. Gu, H. Feng, H. Sun, P. Liu, Q. Yue, J. Hu, C. Cao, and Y. Zhang, "Hierarchical attention network for interpretable and fine-grained vulnerability detection," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2022, pp. 1–6.

[33] D. Zou, Y. Hu, W. Li, Y. Wu, H. Zhao, and H. Jin, "mvulpreter: A multi-granularity vulnerability detection system with interpretations," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[34] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *arXiv preprint arXiv:1909.03496*, 2019.

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

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

[37] S. Pujar, Y. Zheng, L. Buratti, B. Lewis, A. Morari, J. Laredo, K. Postlethwait, and C. Görn, "Varangian: a git bot for augmented static analysis," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 766–767.

[38] S. Mehrpour and T. D. LaToza, "Can static analysis tools find more defects? a qualitative study of design rule violations found by code review," *Empirical Software Engineering*, vol. 28, no. 1, p. 5, 2023.

[39] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD conference*, vol. 5, 2008, pp. 1–20.

[40] K. Umann and Z. Porkoláb, "Detecting uninitialized variables in c++ with the clang static analyzer," *Acta Cybernetica*, vol. 25, no. 4, pp. 923–940, 2022.

[41] P. G. Szécsi, G. Horváth, and Z. Porkoláb, "Improved loop execution modeling in the clang static analyzer," *Acta Cybernetica*, vol. 25, no. 4, pp. 909–921, 2022.

[42] H. Aslanyan, Z. Gevorgyan, R. Mkoyan, H. Movsisyan, V. Sahakyan, and S. Sargsyan, "Static analysis methods for memory leak detection: A survey," in *2022 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2022, pp. 1–6.

[43] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 232–242.

[44] N. S. Harzevili, J. Shin, J. Wang, and S. Wang, "Characterizing and understanding software security vulnerabilities in machine learning libraries," *arXiv preprint arXiv:2203.06502*, 2022.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[51] R. R. Curtin, M. Edel, O. Shrit, S. Agrawal, S. Basak, J. J. Balamuta, R. Birmingham, K. Dutt, D. Eddelbuettel, R. Garg *et al.*, "mlpack 4: a fast, header-only c++ machine learning library," *arXiv preprint arXiv:2302.00820*, 2023.

[52] 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.

[53] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against c and c++ programs," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, pp. 1–28, 2012.

[54] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[55] X. Duan, J. Wu, S. Ji, Z. Rui, T. Luo, M. Yang, and Y. Wu, "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities." in *IJCAI*, 2019, pp. 4665–4671.

[56] 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.

[57] 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.

[58] 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.

[59] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, p. 122–131, apr 2016. [Online]. Available: https://doi.org/10.1145/2902362

[60] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[61] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[62] 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.

[63] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *15th International symposium on software reliability engineering*. IEEE, 2004, pp. 245–256.

[64] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[65] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 241–252.

[66] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.

[67] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008, pp. 329–339.

[68] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 55–64.

[69] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 61–72.