Noname manuscript No.
(will be inserted by the editor)

IoTvulCode - AI-enabled vulnerability detection in software products designed for IoT applications

Guru Prasad Bhandari · Gebremariam Assres · Nikola Gavric · Andrii Shalaginov · Tor-Morten Grønli

Received: date / Accepted: date

Abstract The proliferation of the Internet of Things (IoT) paradigm has ushered in a new era of connectivity and convenience. Consequently, rapid IoT expansion has introduced unprecedented security challenges, among which source code vulnerabilities present a significant risk. Recently, machine learning (ML) has been increasingly used to detect source code vulnerabilities. However, there has been a lack of attention to IoT-specific frameworks, in terms of both tools and datasets. In this paper, we address potential source code vulnerabilities in some of the most commonly used IoT frameworks. Hence, we introduce *IoTvulCode* - a novel framework consisting of a dataset-generating tool, and ML-enabled methods for the detection of source code vulnerabilities and weaknesses as well as the initial release of an IoT vulnerability dataset. Our framework contributes to improving the existing coding practices, leading to a more secure IoT infrastructure. Additionally, *IoTvulCode* provides a solid basis for the IoT research community to further explore the topic.

Keywords Internet of Things · Vulnerabilities Dataset · IoT security · Source Code · Machine Learning · Natural Language Processing

1 Introduction

Internet of Things (IoT) refers to the interconnected physical objects connected to the internet or each other, relying mostly on Wireless Sensor Networks (WSN) to exchange information without human intervention [29]. These devices are widely used in both consumer and industrial applications due to the benefits of the automation they bring. IoT use cases range from smart homes, vehicular technology, and healthcare, to transport, power, and agriculture. In recent years, we have witnessed a rapid increase in the number and complexity of IoT infrastructure. However, IoT devices have posed security threats and vulnerabilities over the years, primarily due to the programming languages used as well as programmers' disregard for secure coding practices [20]. The IoT Operating Systems(OS) and applications are vulnerable to security breaches and the existing higher-level security measures may not help [2].

Open Web Application Security Project (OWASP) [30], Common Vulnerability and Enumerations (CVE) [26], and Common Weakness Enumeration (CWE) [13], and Common Vulnerability Scoring System (CVSS) [12] are major resources for understanding, categorizing and addressing vulnerabilities, including those in IoT systems. These resources can be used to understand the common types of vulnerabilities that affect IoT systems. Particularly, OWASP publishes standard awareness documents for developers and web application security which represent the most critical security risks to various systems as reported IoT top 10 list [30], the information can be considered by IoT developers for more secure coding. There is also a push to incorporate security into DevOps through the development of DevSecOps (Development, Security, and Operations) [1]. DevSecOps concentrates on integrating security controls and practices into the DevOps cycle, offering substantial potential for the development of secure IoT software.

Furthermore, in real-world software systems, many types of IoT data, including, network traffic, sensor readings, metrics, logs, alerts, and traces play an es-

School of Economics, Innovation, and Technology
Address: Kirkegata 24, 0153, Oslo, Norway
Tel: +47 41352384
E-mail: guruprasad.bhandari@kristiania.no

sential role in cybersecurity engineering. In particular, network traffic has been widely exploited for malware and attack detection [4, 16, 31]. Similarly, sensor data is used for anomaly detection and environment-control measures. However, the aforementioned methods are incapable of detecting threats in advance, because they are not designed for a such purpose. When it comes to threat detection, most of the existing security threats originate from the vulnerabilities in the code [6, 25, 38].

The attackers can exploit security vulnerabilities to compromise the affected system's data and functionality as well as possibly use them for further malicious activities. Therefore, static application security testing (SAST) is an important process of the DevSecOps pipeline, during which source code is automatically analyzed to identify security vulnerabilities in the early development stages [1].

In this study, we present *IoTvulCode* - a comprehensive framework consisting of a data extraction tool for *C/C++* source code vulnerabilities, and ML and natural language processing (NLP) methods to detect them. We also provide an initial release of an IoT vulnerability dataset. We collected the source code of the most commonly used IoT projects to create a labeled dataset of both vulnerable and benign samples. To create a generic dataset, we only included projects containing CVE-recorded vulnerable entries. The types of vulnerabilities in the extracted dataset are labeled according to CWE categorization. The main contributions of this study are the following:

- open-source vulnerability dataset extraction tool,
- ML and NLP-based vulnerability detectors, and
- an initial IoT vulnerability dataset.

The remainder of the paper is organized as follows. Section 2 discusses the existing datasets for identifying IoT vulnerable codes and existing AI-based approaches to vulnerability classification of IoT projects. Section 3 represents the details of the proposed *IoTvulCode* methodology. Section 4 elaborates on the initial release of the *IoTvulCode* dataset by the proposed tool and its statistics. Similarly, Section 5 presents the experimental results and comparative performance measures of the ML modules on the extracted dataset. The section also discusses the observations and challenges in AI-based models for vulnerability detection on the source code of IoT projects. Finally, Section 6 concludes the paper.

2 Related work

Together with the challenges confronted by the Internet, IoT faces substantial challenges (including scalability, mobility, and resource limitations) due to a massive number of interconnected devices, and heterogeneity of exchanged data [37]. Researchers and practitioners have delved into various OSI layers to scrutinize security concerns within the IoT software development process, a crucial aspect of the DevSecOps pipeline. In this section, we present an overview of code vulnerability detection and describe prior work most related to our study, construction of the IoT vulnerability dataset, and detection of vulnerability in IoT smart environments.

2.1 Overview of code vulnerability detection

The rising number of security vulnerabilities in software highlights the need for improved detection methods. Literature shows that there is a practice of using automated source code scanning tools, specifically static code analysis, for early detection of vulnerabilities in classic software development. For example, a survey conducted in [23] explores how deep learning and neural network approaches are applied in detecting software vulnerabilities, by leveraging a large amount of open-source code.

Leveraging a large dataset of C and C++ functions, [35] developed a scalable vulnerability detection tool using deep feature representation learning, demonstrating its effectiveness on real software. Similarly, [21] compared three C/C++ tools (flawfinder, RATS, CPPCheck) and two JAVA tools (Spotbugs and PMD) evaluating the categories of vulnerabilities detected and the likelihood of false positives. The authors pointed out variations in detection capabilities and false positive reporting among the tools.

Github [18] also supports building secure code security into the GitHub workflow with features to keep secrets and vulnerabilities in the codebase. Once we commit any changes to the repository, it automatically scans the code for security vulnerabilities. However, the project should be hosted in GitHub to use that automatic code scanning feature. Similarly, Hanif and Maffeis [19] presented *VulBERTa* model which was pretrained on *RoBERTa* [24] model with a custom tokenization pipeline on real-world code from open-source C/C++ projects. Their deep-learning approach to detect security vulnerabilities in source code was evaluated on binary and multi-class vulnerability detection tasks across several vulnerability datasets extracted from general software.

In a related context, [39] introduced another framework for vulnerability detection, namely FUNDED (Flow-sensitive vUl-Nerability coDE Detection), which employs graph neural networks to capture program dependencies and operates on a graph representation of source code, yielding better representations for vulnerability detection. According to the authors, the framework outperforms six state-of-the-art models across various programming languages, showcasing its effectiveness. Additionally, a review was conducted focusing on Android application analysis and source code vulnerability detection methods. It critically assesses both Machine Learning (ML)-based and conventional methods, aiming to guide researchers in enhancing secure mobile application development and minimizing vulnerabilities, particularly through ML approaches.

2.2 IoT vulnerability code dataset

Analyzing datasets containing vulnerable code associated with IoT is vital for advancing our understanding of security issues, identifying prevalent vulnerabilities, and developing effective ML models to enhance the security of IoT software. Al-Boghdady et al. [2] have created a tool called iDetect for detecting vulnerabilities in the C/C++ source code of IoT operating systems (IoT OSs). The labeling of the dataset was done using static code analyzing tools (SATs) - Cppcheck version 2.1 [11], Flawfinder version 2.0.11 [14], and Rough Auditing Tool for Security (RATS) [32]. Alnaeli et al. [3] conducted an empirical study involving 18 open-source systems, encompassing millions of lines of C/C++ code utilized in IoT devices. Static code analysis methods were employed for each source code project to identify unsafe commands, such as *strcpy, strcmp, and strlen*, which pose potential risks to the system. Celik et al.[7] introduced an IoT-specific test suite, *IoTBench*, an open-source repository for evaluating information leakage in IoT apps. The *IoTBench* includes 19 hand-crafted malicious SmartThings apps that contain 27 data leaks via either Internet or messaging service sinks.

Some of the generic datasets for vulnerability detection that are publicly available are summarized in Table 1. The *iDetect* dataset is the only IoT code-specific dataset. However, the dataset has just included 6,245 samples (3,082 vulnerable and 3,163 non-vulnerable) after removing the duplicates and ambiguous samples. In comparison with *iDetect*, our *IoTvulCode* dataset is 162.4 times bigger than *iDetect* covering 1,014,548 statements (948,996 benign and 65,052 vulnerable samples).

The *IoTvulCode* dataset and the extraction tool are unique to existing studies in several ways as follows-

- The size of the dataset is bigger in terms of the sample size than the existing IoT-specific source-code dataset.
- The open-source extraction tool makes it easy to add new projects to the list to crawl more data.
- The incremental service of the tool facilitates the stop and resume feature to the projects in case the system hangs in the intermediate state.
- The dataset includes binary and multi-class vulnerability types which enable the classification of binary as well as multi-class.
- Specially, existing commit-based datasets may suffer accuracy because they assume all the changes made in the commit were vulnerable code, however, our tool picks the exact occurrences of the vulnerable code rather assumption based vulnerable code.

2.3 ML models for IoT code vulnerability detection

Most of the existing studies on IoT security systems have concentrated on pinpointing security issues associated with IoT communication processes, data privacy, and authentication methods. Naeem and Alalfi [27] have presented deep learning-based vulnerability identification of IoT system applications. The method categorizes the vulnerabilities that lead to sensitive information leakage that can be identified using taint flow analysis on synthesized test-suite dataset [7]. The source code is converted into a list of tokens and then transformed into vectors (token2vec). Additionally, the identified tainted flows are also transformed into vectors (flow2vec). Similarly, Nazzal and Alalfi have proposed a tainted flow static analysis approach for the identification and reporting of information leakage in the Smarthings IoT app.

Gao et al. [17] have presented *IoTSeeker* a function semantic learning-based vulnerability search approach for cross-platform IoT binary. The *IoTSeeker* aims to design a vulnerability search approach using semantic feature extraction and a neural network that can automatically inform whether a given binary program from IoT devices contains clone vulnerabilities or not.

3 Methodology

The CVE provides vulnerability records of all the software and hardware systems and releases them publicly with their references. There are more than 122,000 vulnerability entries in the CVE database. Some of the vulnerabilities occur within the C/C++ source-code function and provide corresponding source-code references. We analyze the CVE references to select only the IoT-related ones. The intuition behind this is that IoT-related code vulnerability follows similar characteristics on different systems. Natural language processing

Table 1: The publicly available generic datasets for vulnerability detection

dataset	granularity	labeling	label accuracy	publication year
VulDeePecker [22]	slice	semi-synthetic	High	2018
Draper [34]	function	Static Analyzer Category filter	low	2018
Devign [40]	Function	Manual	high	2019
CVEfixes [5]	Commit	Security Issues	High	2021
DiverseVul [9]	Commit	Security issues	High	2023
ReVeal [8]	Function	Security Issues	High	2021
BigVul [15]	Commit	Security Issues	High	2020
CrossVul [28]	Commit	Security Issues	High	2021
iDetect [2]	Statement	Static Analyzer	low	2022

(NLP) or machine learning-based approach will be useful to detect such patterns.

3.1 Static security analysis tools

Some static analyzers use similar techniques to detect security bugs and abnormal behavior of the codes and some use unique techniques. The use of multiple analyzers covers multiple weaknesses of the source code than a single analyzer. Therefore, in this study, we have used three static analysis tools; *FlawFinder, CppCheck*, and *Rats*.

FlawFinder: This static analyzer is licensed under GNU GPLv2. For the data extraction, we used *FlawFinder* [14] version 2.0.19 (⋆ 376) which was released on Aug 29, 2021. The tool implements a syntactic analysis technique to scan C/C++ source code for potentially vulnerable code patterns stored in a local database. It identifies the susceptible vulnerabilities at the function level from the integrated rules in the past. It also assesses their risk of triggering a security bug by analyzing the arguments in the code, ranking them as likely severity.

CppCheck: It is a static security analysis tool for C/C++ code [11]. The tool is released under GPL3.0 and has obtained 4.9k GitHub ⋆stars. The tool detects bugs and focuses on detecting undefined behavior and dangerous coding constructs. It uses unsound flow-sensitive analysis, unlike other analyzers that use path-sensitive analysis.

Rats: The rough auditing tool for security (*Rats*) [33] is an open-source tool licensed under GPL-2.0 developed by Secure Software Inc. The tool scans code of multiple programming languages; C, C++, Perl, PHP, Python, and Ruby. Unlike other tools, *Rats* performs only a rough analysis of source code flagging common security-related errors such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions.

3.2 Other supporting tools

In addition to the above static security analysis tools, we have used the following libraries and tools for the construction of the *IoTvulCode* dataset-

srcML: this is a software tool for the exploration, analysis, and manipulation of source code [10]. The tool is mainly used to convert source code into abstract syntax tree (AST) and back into source code, which allows converting source code into language-independent format (XML) and translating code of one programming to another. In this study, we have used *srcML* tool to retrieve source code into function blocks and perform *srcML* transformation, in the following order: *code* → *AST* → *function blocks* → *function code.*

Guesslang: this is an open-source tool used to recognize the programming language of the source code file [36]. The tool is trained using deep learning methods with over a million source-code files and supports 54 programming languages. The *Guesslang* tool is precise to guess the language with guessing accuracy higher than 90%, however, it takes considerable time to guess.

3.3 The *IoTvulCode* dataset extraction method

The referred IoT software are crawled and analyzed for security vulnerabilities and flaws using static code analysis tools, *FlawFinder*, *CppCheck*, and *Rats.* Once the security flaws contexts were extracted from the tools, the corresponding file of the project was analyzed to extract the statement-level, and function-level metrics to provide additional code information in the dataset. The extracted metrics include the actual vulnerable code statements, corresponding function blocks, function metrics, file names, project names, vulnerability labels, and some additional information. Algorithm 1 summarizes all the major steps of the data extraction pipeline. Additionally, Figure 1 also shows the proposed extraction framework for the collection of vulnerability data which is also briefly as follows:

3.3.1 Vulnerable samples extraction

The vulnerable sample extraction mainly involves scanning the projects and composing the collected vulnerable data into an *SQLite2* database file. It resembles from step 1 to step 4 as shown in Figure 1.

1. To extract the vulnerable sample, at first, the source code of the projects should be crawled locally and their directories listed in the configuration file (*ext_ projects.yaml*). The user can provide the initial input parameters, i.e., database name and other settings. The iteration of the extraction process goes into each project on an incremental basis. If the status of any of the projects is 'Not Started' or 'In Progress' then the extraction continues for the remaining files of each incomplete project.
2. This step (optional) checks whether the project is registered to CVE vulnerability records or not. This process is mainly to pick only the benchmarked IoT software. The notion is that the project registered in the CVE records follows standard coding practices.
3. This step scans project files and applies the static security analysis tools. For the projects that were incomplete in scanning the vulnerable data, only the remaining files will be extracted ignoring the already stored files in the database. The user can select either *guesslang* or file extension-based method options to classify the programming language of the file because the file-extension-based method is very fast as compared to the *Guesslang.* If a file extension is not in the given programming language list, then it is set to 'unknown'. On each file, the static analyzers run for the detection of vulnerability and weakness. The analyzers retrieve the composed results of the statement-level vulnerability data of the file.
4. The next step is to compose the generated vulnerable statements and populate the function-level data from the statements. In this study, we have used *srcML* [10] for fetching the function-level data.

3.3.2 Benign sample extraction

To apply machine learning techniques to vulnerability assessment, we require both vulnerable and benign (non-vulnerable) samples. The given static analyzers only provide the context or line of the vulnerable code in the file and its line number. We have carried out several steps to collect benign samples for both statement-level and function-level data.

5. The function is labeled as *vulnerable* if it contains any of the vulnerable statements resulting from static analyzers on the file. The rest of the functions of the file are labeled as *benign* samples.
6. For gathering the *benign* statements, we took randomly sampled non-vulnerable statements from the function bodies of the file.

Algorithm 1: Extraction of IoT vulnerable data

```
Data: Links of the IoT project directories stored
      locally
Result: Vulnerability data of IoT projects
1  initialization;
2  while each IoT project do
3      check the entry in CVE records ;
4      if project not in CVE then
5          ignore the project ;
6      else
7          while each file of the project do
8              apply analyzers; CppCheck, FlawFinder,
                 and Rats;
9              if statement is 'vulnerable' then
10                 label statement as 'vulnerable' with
                     CWE type;
11                 store the information and the label ;
12             else
13                 label 'code-block' as 'benign';
14                 store the information and the label ;
15             end
16             while check each functions do
17                 if statement is in function then
18                     label function as 'vulnerable' ;
19                     store the information and label ;
20                 else
21                     label function as 'benign' ;
22                     store the information and label ;
23                 end
24             end
25         end
26         compose vulnerable data of all files of the
             project ;
27     end
28 end
29 compose vulnerable data of all projects ;
30 filtering the duplicates ;
31 removing ambiguous samples ;
32 construct the refined database ;
```

3.4 Vulnerability detection framework

Creating an ML model for vulnerability detection involves several steps of MLOps. The high-level overview of the steps is presented in Figure 2 and Algorithm 2 also explained as follows-

1. *Data collection:* The above data extraction process gives us a dataset of IoT software both vulnerable

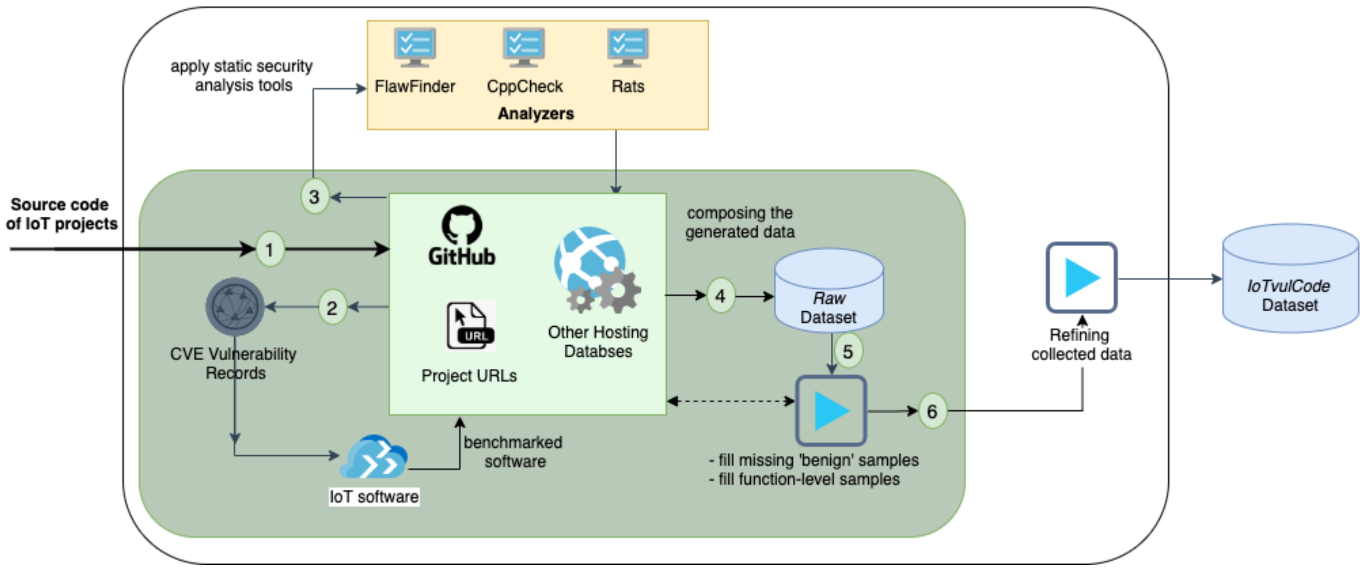

apply static security analysis tools
FlawFinder
CppCheck
Rats
Analyzers
Source code of IoT projects
GitHub
Other Hosting Databses
Project URLs
composing the generated data
Raw Dataset
Refining collected data
IoTvulCode Dataset
CVE Vulnerability Records
IoT software
benchmarked software
- fill missing 'benign' samples
- fill function-level samples
1
2
3
4
5
6

Fig. 1: The proposed framework for vulnerability data collection

and non-vulnerable. This could be from open-source projects, vulnerability databases, and other sources.

2. *Preprocessing:* Perform several preprocessing steps to the code to convert it into a format suitable for ML models. In code analysis, this could involve parsing raw code, tokenization, and vectorization to represent code into an encoded sequence as shown in step-1 in Figure 2.
3. *Model training*: Train an ML model on the preprocessed data for the detection of vulnerabilities involved as a next step (step 2 of Figure 2. The training process implements sequence models like RNNs or LSTMs as involved in capturing the sequential nature of the code.
4. *Evaluation*: This process involves the cross-checking of the trained model on separate data that were not used for the training. Evaluate the model's performance and calculate metrics such as accuracy, precision, recall, and loss (step 3 of Figure 2.
5. *Deployment*: If the model's performance is satisfactory with training and testing, deploy it to a production environment where it can analyze new IoT code for vulnerabilities. The model can be deployed as a plugin in any integrated development environment (IDE) to automatically detect vulnerabilities.

Algorithm 2: vulnerability detection in IoT code

```
Data: labeled dataset of IoT code (from Algo 1)
Result: detection result; vulnerable or not, or types
        of vulnerability
1  initialization;
2  while preprocessing steps do
3      apply data filtering;
4      remove duplicate samples ;
5      remove ambiguous samples ;
6      tokenized the input code ;
7      vectorized the sequence ;
8  end
9  split data into train/test sets;
10 while ML sequence models do
11     apply the ML models ;
12     if perform train then
13         train ML model;
14     else
15         examine the trained ML model;
16     end
17 end
```

3.5 Experimental setup

The resource-intensive operations, i.e. training of the machine learning models were carried out on NVIDIA DGX (DualProcessor Intel Xeon Scalable Platinum 8176 w/ 16 qty Nvidia Volta V100) and NVIDIA HGX (DualProcessor AMD EPYC Milan 7763 64-core w/ 8 qty Nvidia Volta A100/80GB). Both high-performance infrastructures (HCI) have GPU power for parallel executions which is suitable for neural network matrix multiplications. Both the infrastructures are hosted in eX3 cluster[1] at Simula Research Laboratory. For the extraction of the dataset, we have used a general-purpose PC - Lenovo Legion 7 powered by AMD Ryzen 7 5800H/3.2 GHz, 16GB RAM, 1TB SSD, and RTX 3080 16GB

[1] https://www.ex3.simula.no/

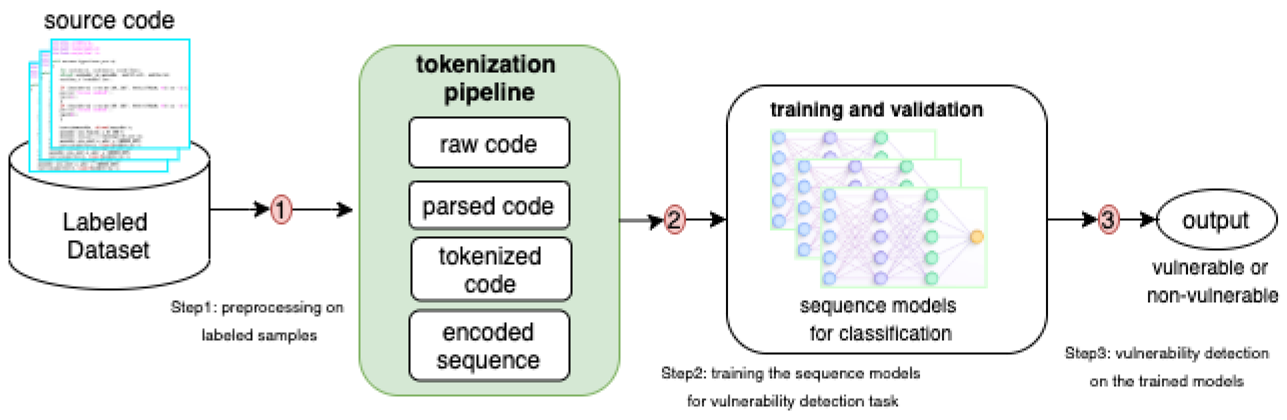

source code
Labeled Dataset
1
Step1: preprocessing on labeled samples
tokenization pipeline
raw code
parsed code
tokenized code
encoded sequence
2
training and validation
sequence models for classification
Step2: training the sequence models for vulnerability detection task
3
output
vulnerable or non-vulnerable
Step3: vulnerability detection on the trained models

Fig. 2: The proposed method for vulnerability detection in IoT OSs and applications

GPU. After downloading all the projects to our local machine, it took 23 hours to extract the vulnerability data from the 11 downloaded projects.

3.6 Hyperparameter settings

RNN and LSTM models training and testing were carried out, with different setup hyperparameters for the experiment as presented in Table-2. Additionally, we have used *categorical_crossentropy* for multiclass and *binary_crossentropy* for binary classification on both statement and function level data.

4 The *IoTvulCode* dataset

The *IoTvulCode* dataset is constructed from the source code of the IoT projects, which are listed in Table 3, along with their versions and links to the open-source repositories. The projects are selected based on the following criteria: (1) the project is an IoT project (OS or software), (2) the project is open-source, (3) the project is written in C/C++, (4) the project is actively maintained, and (5) the project is popular (checking CVE records).

Table 2: Hyperparameter settings for training sequence models

hyperparameters	statement	function
epochs	120	120
batch	128	128
input_length	150	1024
input_dim	150	1024
output_dim	1 or more	1 or more
lr	1e-4	1e-4
patience	35	35
optimizer	adam	adam
l2_reg	1e-4	1e-4
dropout	0.002	0.002
recur_dropout	0.002	0.002
beta_1	0.09	0.09
beta_2	0.099	0.099
epsilon	1e-08	1e-08
decay	0.0	0.0

Table 3: List of the IoT projects

Project	version	URL
linux-rpi	6.1.y	www.raspberrypi.com/software/
ARMmbed	6.17.0	https://os.mbed.com/mbed-os/
FreeRTOS	202212.01	www.freertos.org/a00104.html
RIOT	2023.07	https://github.com/RIOT-OS/RIOT
contiki	2.4	https://github.com/contiki-os/contiki
gnucobol	3.2	https://gnucobol.sourceforge.io/
mbed-os	6.17.0	https://github.com/ARMmbed/mbed-os
miropython	1.12.0	https://micropython.org/
mosquito	2.0.18	https://github.com/eclipse/mosquitto
openwrt	23.05.2	https://github.com/openwrt/openwrt

4.1 Dataset overview

In the current version of the extracted dataset, there are 1,014,548 statements (948,996 benign and 65,052 vulnerable samples) and 548,089 functions (481,390 benign and 66,699 vulnerable samples). Among all extracted projects, *linux-rpi* has the most recorded entries with 816,672 total statements and 456,380 functions, which is followed by *ARMmbed* with 43,782 statements and 26,095 functions. Of course, the severity of the project can be seen in the size of the vulnerability and weakness samples present in the project. However, *linux-rpi* being the biggest project in the list can tend to hold a higher number of vulnerable samples. Table 4 shows further detailed information on the frequency of the vulnerable and benign samples in both the statement- and function of all the extracted projects.

project	#statement		#function	
	Benign	Vulnerable	Benign	Vulnerable
ARMmbed	37,798	5,984	21,206	4,889
FreeRTOS	37,980	4,398	17,196	3,675
RIOT	13,349	2,357	7,583	2,227
contiki	3,756	977	1,730	568
gnucobol	8,828	1,356	1,941	909
linux-rpi	772,050	44,622	407,684	48,696
mbed-os	27,305	18	216	22
micropython	35,440	3,662	19,005	4,126
mosquitto	4,139	490	817	558
openwrt	6,519	861	3,068	677
tinyos	2,249	410	944	352
TOTAL	949,413	65,135	481,390	66,699

Table 4: Number of statements and function on the extracted projects

4.2 Major vulnerabilities and weaknesses

The majority of the static analyzers categorize the vulnerabilities and weaknesses based on CWE type, which is used as a labeling technique for multiclass vulnerability identification. The Figure-3 sunburst plot (or multi-level pie chart) visualizes the hierarchical data structures of the vulnerability and weakness type, i.e., frequency of the CWE category, name, and types. In the figure, the majority of classes- memcpy of CWE-120 type (#21153 samples) and char of CWE-119/-120 type (#16396 samples) covered more than half of the vulnerability samples.

More specifically, the top 10 CWEs in the statement- and function-level data are shown in Table 5. At the statement level, CWE-120 (Buffer Copy without Checking Size of Input) is the most frequent CWE with 30,953 samples, followed by CWE-119!/CWE-120 (Improper Restriction of Operations within the Bounds of a Memory Buffer) with 16408 samples. In function-level, again CWE-120 is the most frequent CWE with 28,119 samples, followed by CWE-119!/CWE-120 with 12,014 samples.

4.3 Sequences sizes of the source-code

Sequence models, such as RNN, LSTM, and transformers are the most popular models for the NLP-based classification and translation of the code. To plan for the correct sequence size of the NLP-based models for the prediction of vulnerabilities and weaknesses, observation of the common distribution of the token sizes is important. Therefore, Figure 4 and Figure 5 show the distribution of the number of tokens in the statement and function-level source code respectively. Each sequence of the input data can be padded to the vocabulary size. The vocabulary size of the NLP-based models is the number of unique words in the dataset.

Similarly, Figure 6 shows the frequency of the number of characters in the statement-level data. The majority of the statements have 10 to 80 characters, and the average number of characters in a statement is 38. The number of characters in a statement is a good indicator of the vocabulary size of the NLP-based models.

Table 5: Top 10 CWEs in statement- and function-level data

cwe	#statements	cwe	#functions
Benign	949,413	Benign	481,390
CWE-120	30,953	CWE-120	28,119
CWE-119!/CWE-120	16,408	CWE-119!/CWE-120	12,014
CWE-126	4,630	CWE-126	5,503
CWE-190	2,303	CWE-unknown	5,008
CWE-120, CWE-20	2,256	CWE-457	4,785
CWE-362	1,794	CWE-190	2,550
CWE-134	1,689	CWE-120, CWE-20	2,496
CWE-457	1,648	CWE-362	2,277
CWE-362/CWE-367!	598	CWE-134	1,116

5 Experimental results

The dataset needs benchmarking to check whether machine learning models, especially NLP-based approaches to the data perform well with the prediction of vulnerabilities and weaknesses in both statement and function-level data. Sequence models, such as recurrent neural networks (RNN) and long-short-term memory networks (LSTM), are well-suited for classification problems for the detection of vulnerability in code because they are designed to work with sequential data. RNN model processes sequences of code by maintaining a hidden state that captures information about the tokens so far. However, it vanishing gradient problem capturing long-term dependencies. LSTM overcomes this using an explicit memory cell that allows them to capture long-term dependencies and makes them more effective for tasks like vulnerability detection, where context from earlier in the code can be important for identifying vulnerabilities. For vulnerability detection, these ML models should be trained on both vulnerable and benign samples for binary classification. For multiclass classification, they should be labeled as vulnerability types (i.e. CWE types).

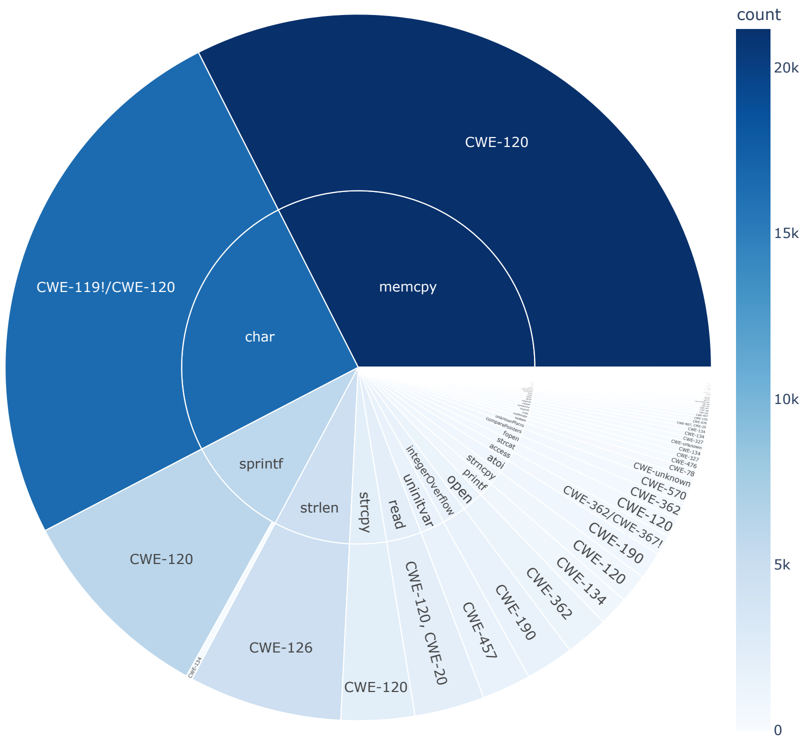


Fig. 3: The sunburst chart showing the frequency of vulnerability categories, names, and CWEs

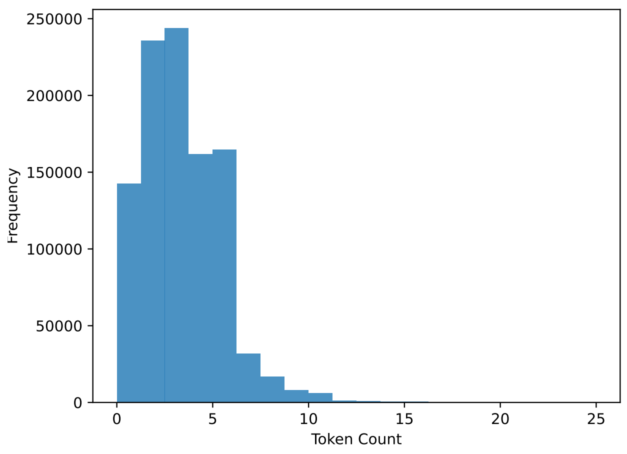


Fig. 4: The frequency of #tokens in the statement-level data

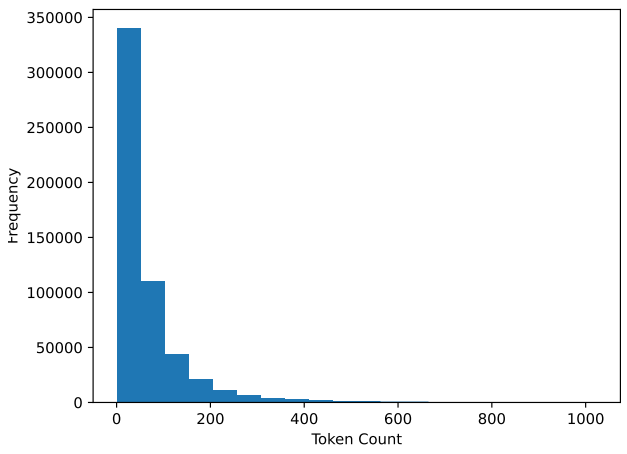


Fig. 5: The frequency of #tokens in the function-level data

5.1 Performance of the models

The performance scores of the ML models provide insights into how well each model is performing in terms

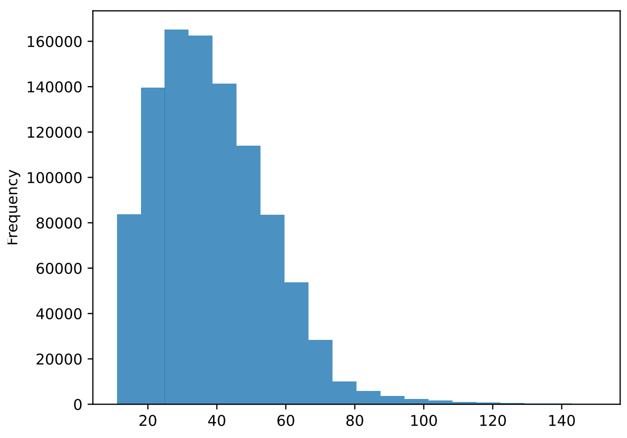


Fig. 6: The frequency of #chars in the statement-level data

of classifying statements in the *IoTvulCode* dataset, with a focus on detecting vulnerabilities. The performance metrics help assess the model's overall accuracy and its ability to correctly identify positive instances while minimizing false positives and false negatives.

The loss in training and validation on the dataset over time with multiple ML models are as given in Figure-7. The loss curve (also known as the learning curve) shows the loss function value as a function of the number of training epochs. At the beginning of training, the loss is typically high meaning the model has not learned anything yet. As iteration increases, the loss should decrease to indicate the model is learning the data to predict the target variable more accurately. In our experiment, in both training and testing causes, the decreasing loss curve tends to 0.01 showing that the model is learning well to predict the vulnerabilities accurately.

Similarly, the accuracy scores of training and validation on our *IoTvulCode* dataset using multiple ML models are projected in Figure 8. An accuracy curve visualizes the accuracy of a model over training epochs. The increasing accuracy over time indicates the model is learning to predict the target label more accurately. Being both training and validation accuracy almost similar in the plot indicates there is no overfitting, i.e. it is able to generalize the data.

Along with accuracy, precision, and recall are also two fundamental metrics used to evaluate the performance of ML models, especially in classification problems such as vulnerability detection in source code. Precision is the ratio of correctly predicted positive observations (for example, vulnerable samples) to the total predicted positive observations. Whereas recall is the ratio of correctly classified positive observations to all observations in the actual class. Figure-9 and Figure-10 show the training and validation precision and recall, trained on our statement-level data using different ML models.

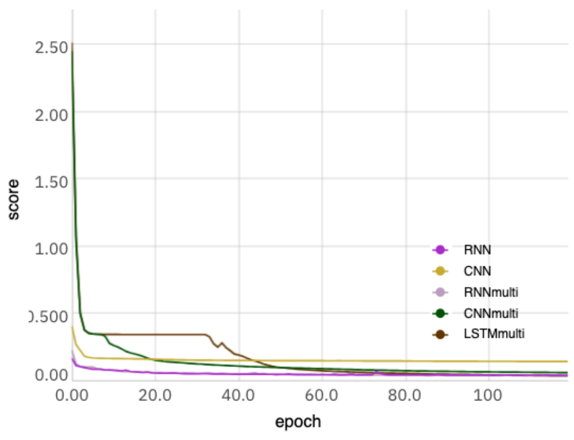
Table 6 provides a summary of performance scores of different ML models on the *IoTvulCode* dataset at the statement level for both training and validation sets. The table scores loss, accuracy, precision, and recall scores of both binary classifications(*IoTvulCode -RNN, CNN, and iDetect-RNN, -CNN*) and multiclass classification (*IoTvulCode -RNNmul, -CNNmul, -LSTMmul*).

For binary classification, the calculated scores indicate that the *IoTvulCode -RNN* model achieves superior results in both the training and validation sets of the *IoTvulCode* dataset, boasting an accuracy score of 0.99 and a precision score of 0.99. Specifically, the training recall stands at 0.97, while the validation recall reaches an even higher value of 0.99. Our ML models do better than the *iDetect* classifiers, having a lower loss score (0.044) as compared to *iDetect* (lowest loss 0.196 training and 0.236 validation in RNN) and better performance in most measures except recall. Our *IoTvulCode* dataset is much larger than *iDetect*, being 162.4 times bigger and including 1,014,548 unique statements (948,996 benign and 65,052 vulnerable samples). Even though the *iDetect* has a higher recall, the dataset may lack performance in the general scenarios of IoT software.

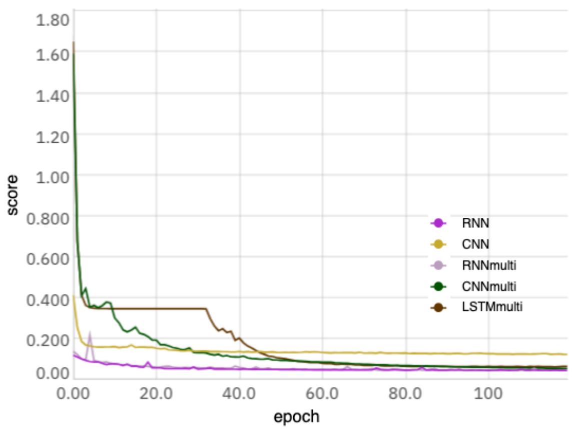
For multiclass classification, the calculated scores indicate that the *IoTvulCode -RNN* model achieves superior results in both the training and validation sets of the *IoTvulCode* dataset, boasting an accuracy, precision, and recall score of 0.99, among all three multiclass classifiers (*IoTvulCode -RNNmul, -CNNmul, LSTMmul*). Comparing the loss score, *IoTvulCode -RNNmul* performs better in the training set and *IoTvulCode -LSTMmul* performs better in the validation set. In the case of multi-classification, pinpointing the precise multiple labels in the *iDetect* dataset proved challenging. Additionally, their dataset contains numerous duplicate and ambiguous samples, necessitating additional preprocessing.

5.2 Discussion on the proposed *IoTvulCode* method

For application-level software testing, bad coding practices not only leave the code vague to understand but also leaves loopholes and weakness in the code. The identification of the vulnerabilities in the early stages of the software development life cycle helps reduce the cost of maintenance and ensures the program is more

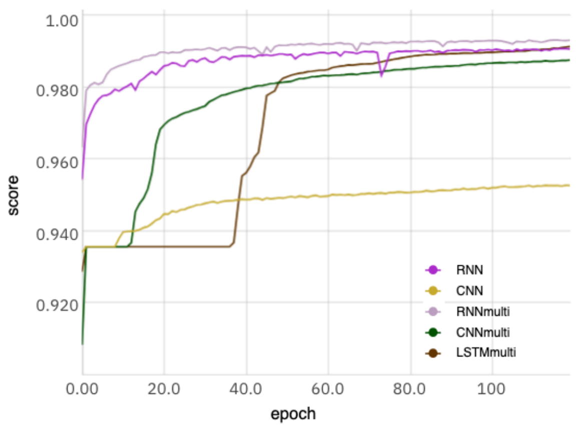


(a) training

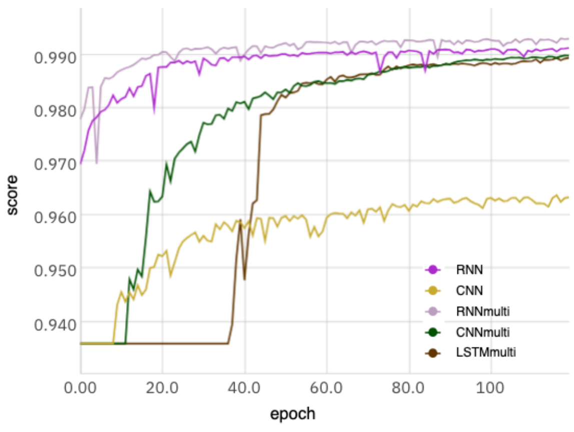


(b) validation

Fig. 7: Training and validation loss on the *IoTvulCode* dataset with different ML models

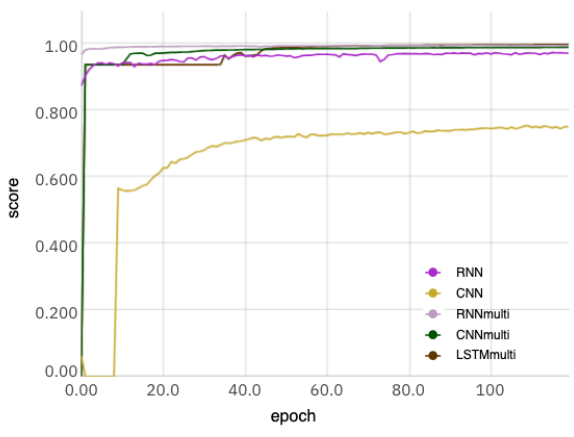


(a) training

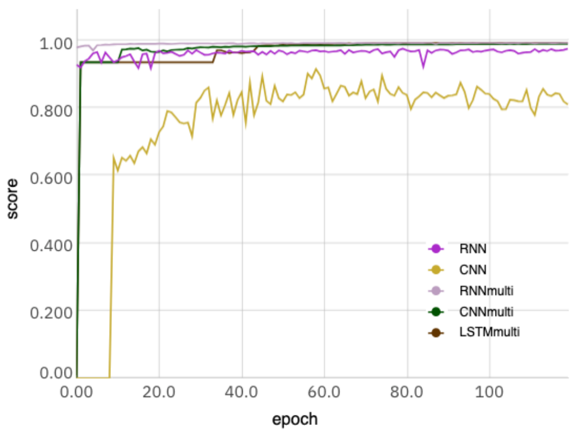


(b) validation

Fig. 8: Training and validation accuracy on the *IoTvulCode* dataset with different ML models

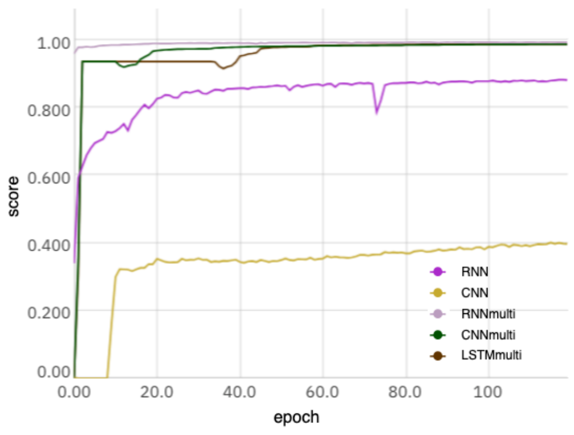


(a) training

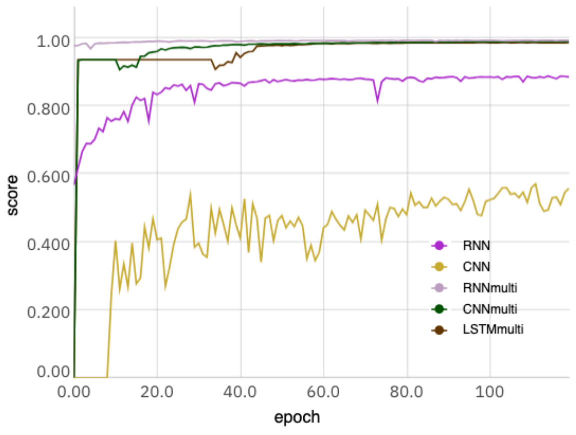


(b) validation

Fig. 9: Training and validation precision on *IoTvulCode* dataset using different ML models



(a) training



(b) validation

Fig. 10: Training and validation recall on *IoTvulCode* dataset using different ML models

Table 6: Summary of the performance scores at statement-level *IoTvulCode* dataset

model	***set***	***loss***	***acc***	***precision***	***recall***
IoTvulCode -RNN	train	0.044	0.991	0.97	0.881
	val	0.045	0.991	0.99	0.885
IoTvulCode -CNN	train	0.147	0.953	0.75	0.397
	val	0.123	0.962	0.96	0.557
iDetect-RNN	train	0.196	0.928	0.939	0.951
	val	0.236	0.926	0.928	0.959
iDetect-CNN	train	0.437	0.928	0.928	0.944
	val	0.436	0.921	0.946	0.929
IoTvulCode -RNNmul	train	0.044	0.99	0.99	0.992
	val	0.045	0.99	0.99	0.991
IoTvulCode -CNNmul	train	0.064	0.987	0.987	0.986
	val	0.057	0.989	0.989	0.989
IoTvulCode -LSTMmul	train	0.042	0.991	0.996	0.987
	val	0.062	0.988	0.993	0.985

secure and robust. The proposed *IoTvulCode* extraction tool [2] and the initial version of the dataset can be utilized in multiple applications for the assessment of IoT vulnerabilities in source code-

- The *IoTvulCode* extraction tool can easily be extended for other applications not only limited to the IoT software but also to the generic software.
- The initial release of the *IoTvulCode* dataset `https://github.com/SmartSecLab/IoTvulCode` can be utilized for the detection of a vulnerability to check its presence in the source code of the IoT software.
- Similarly, the labeling of the dataset is based on CWE weakness types, which supports the multiclass prediction of the vulnerability, not only the presence but also the category of vulnerability that appeared in the code.
- The extracted dataset by the *IoTvulCode* tool also has multiple granular levels of source code snippets; statement-level and function-level. Therefore the dataset enables vulnerability assessment at a multi-granular level.
- The dataset and its extraction tool are open-source licensed which enables the interested user to replicate, extend, and redistribute the tool and the extracted dataset.

The extraction tool, the initial release of the dataset, and the ML models will open up the research in the implementation of NLP and machine learning models for the detection of vulnerabilities and security flaws in IoT source code at both statement-, and function-levels.

6 Conclusion

The detection of vulnerabilities and weaknesses in IoT operating systems and applications is a critical aspect of ensuring the security and reliability of interconnected devices in the smart world. As a component of the DevSecOps pipeline for vulnerability detection, our proposed tool scans the source code of the IoT software and identifies possible loopholes in the source code. In this study, we created a dataset, named *IoTvulCode* , which is labeled as binary and multiclass, based on CWE's most common IoT code vulnerabilities. The dataset contains around a million statements (6.5% vulnerable samples) and around half a million functions (12% vulnerable samples).

Additionally, we applied several ML methods and trained the models to detect vulnerabilities in the C/C++ source code of IoT software and compare and validate the models. Our experiment shows that the RNN model

[2] https://github.com/SmartSecLab/IoTvulCode

achieved a binary accuracy of 99%, precision of 97%, recall of 88%, and a multiclass accuracy, precision, and recall of 99% on the labeled *IoTvulCode* dataset. In future work, we will extend the labeled dataset to cover other generic software projects and also to identify security issues. We will exploit more sequence models and transformers to fine-tune the existing models like *VulBERTa* [19] which better understand the semantics of the code hence improving the performance.

Data availability

The experimented models including the source code of the study are publicly available at the GitHub repository- `https://github.com/SmartSecLab/IoTvulCode`. The initial version of the extracted *IoTvulCode* dataset is available at zenodo with assigned DOI-`DOI:10.5281/zenodo.10203899` (To facilitate the review process, we provide a zipped file of the project repository attached along with the manuscript, which is also available at Dropbox link - `https://www.dropbox.com/scl/fi/es7mzxgmvi6mjx9lka4cp/IoTvulCode.zip?rlkey=jucux3h6kv7zzdqan7q2fnkkj&dl=0`). The plots and figures presented in the paper can be reproduced running the jupyter notebook available at `https://github.com/SmartSecLab/IoTvulCode/blob/main/notebooks/statistics.ipynb`. We welcome the IoT software security community to reproduce our results and enhance further the detection methods of vulnerabilities and weaknesses of IoT open-source software.

Competing Interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Compliance with Ethical Standards

This material is the authors' own original work, which has not been previously published elsewhere. The paper reflects the authors' own research and analysis in a truthful and complete manner.

Acknowledgment

The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

1. Akula, B.S.: Vulnerability Management in DevSecOps. https://dzone.com/articles/vulnerability-management-in-devsecops (2023)
2. Al-Boghdady, A., El-Ramly, M., Wassif, K.: iDetect for vulnerability detection in internet of things operating systems using machine learning. Scientific Reports **12**(1), 17086 (2022). DOI 10.1038/s41598-022-21325-x
3. Alnaeli, S.M., Sarnowski, M., Aman, M.S., Abdelgawad, A., Yelamarthi, K.: Source Code Vulnerabilities in IoT Software Systems. Advances in Science, Technology and Engineering Systems Journal **2**(3), 1502–1507 (2017). DOI 10.25046/aj0203188
4. Bhandari, G., Lyth, A., Shalaginov, A., Grønli, T.M.: Distributed Deep Neural-Network-Based Middleware for Cyber-Attacks Detection in Smart IoT Ecosystem: A Novel Framework and Performance Evaluation Approach. Electronics **12**(2), 298 (2023). DOI 10.3390/electronics12020298
5. Bhandari, G., Naseer, A., Moonen, L.: CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2021, pp. 30–39. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3475960.3475985
6. Blinowski, G.J., Piotrowski, P.: CVE Based Classification of Vulnerable IoT Systems. In: W. Zamojski, J. Mazurkiewicz, J. Sugier, T. Walkowiak, J. Kacprzyk (eds.) Theory and Applications of Dependable Computer Systems, Advances in Intelligent Systems and Computing, pp. 82–93. Springer International Publishing, Cham (2020). DOI 10.1007/978-3-030-48256-5_9
7. Celik, Z.B., Babun, L., Sikder, A.K., Aksu, H., Tan, G., McDaniel, P., Uluagac, A.S.: Sensitive Information Tracking in Commodity IoT. 27th USENIX Security Symposium (2018)
8. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep Learning based Vulnerability Detection: Are We There Yet. IEEE Transactions on Software Engineering pp. 1–1 (2021). DOI 10/gk52qr
9. Chen, Y., Ding, Z., Chen, X., Wagner, D.: DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection (2023). DOI 10.48550/arXiv.2304.00409
10. Collard, M.L., Decker, M.J., Maletic, J.I.: srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. In: 2013 IEEE International Conference

on Software Maintenance, pp. 516–519 (2013). DOI 10.1109/ICSM.2013.85
11. Cppcheck2.1: A Tool for Static C/C++ Code Analysis. https://cppcheck.sourceforge.io/ (2021)
12. CVSS: NVD - Vulnerability Metrics. https://nvd.nist.gov/vuln-metrics/cvss (2022)
13. CWE: CWE - Common Weakness Enumeration. https://cwe.mitre.org/index.html (2023)
14. dwheeler: Flawfinder v. 2.0.11. https://dwheeler.com/flawfinder/ (2021)
15. Fan, J., Li, L., Wang, S., Nguyen, T.N.: A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In: International Conference on Mining Software Repositories (MSR), p. 5 (2020). DOI 10/gjscq5
16. Ferrag, M.A., Friha, O., Hamouda, D., Maglaras, L., Janicke, H.: Edge-IIoTset: A New Comprehensive Realistic Cyber Security Dataset of IoT and IIoT Applications for Centralized and Federated Learning. IEEE Access **10** (2022)
17. Gao, J., Yang, X., Jiang, Y., Song, H., Choo, K.K.R., Sun, J.: Semantic Learning Based Cross-Platform Binary Vulnerability Search For IoT Devices. IEEE Transactions on Industrial Informatics **17**(2), 971–979 (2021). DOI 10.1109/TII.2019.2947432
18. GitHub: Code security documentation. https://docs.github.com/code-security (2023)
19. Hanif, H., Maffeis, S.: VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In: 2022 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2022). DOI 10.1109/IJCNN55064.2022.9892280
20. Ibrahim, A., El-Ramly, M., Badr, A.: Beware of the Vulnerability! How Vulnerable are GitHub's Most Popular PHP Applications? In: 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA), pp. 1–7 (2019). DOI 10.1109/AICCSA47632.2019.9035265
21. Kaur, A., Nayyar, R.: A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. Procedia Computer Science **171**, 2023–2029 (2020). DOI 10.1016/j.procs.2020.04.217
22. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In: Network and Distributed System Security Symposium (2018). DOI 10/gf96vp
23. Lin, G., Wen, S., Han, Q.L., Zhang, J., Xiang, Y.: Software Vulnerability Detection Using Deep Neural Networks: A Survey. Proceedings of the IEEE **108**(10), 1825–1848 (2020). DOI 10.1109/JPROC.2020.2993293
24. Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V.: RoBERTa: A Robustly Optimized BERT Pretraining Approach (2019). DOI 10.48550/arXiv.1907.11692
25. McLean, R.K.: Comparing Static Security Analysis Tools Using Open Source Software | IEEE Conference Publication | IEEE Xplore. In: 2012 IEEE Sixth International Conference on Software Security and Reliability Companion. IEEE, Gaithersburg, MD, USA (2012). DOI 10.1109/SERE-C.2012.16
26. MITRE: Common Vulnerability and Enumerations (CVE). https://cve.mitre.org/index.html (2023)
27. Naeem, H., Alalfi, M.H.: Identifying Vulnerable IoT Applications using Deep Learning. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 582–586. IEEE, London, ON, Canada (2020). DOI 10.1109/SANER48275.2020.9054817
28. Nikitopoulos, G., Dritsa, K., Louridas, P., Mitropoulos, D.: CrossVul: A cross-language vulnerability dataset with commit data. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1565–1569. ACM, New York, NY, USA (2021). DOI 10/gmvfdq
29. Oracle: What is the Internet of Things (IoT)? https://www.oracle.com/internet-of-things/what-is-iot/ (2023)
30. OWASP: OWASP Internet of Things | OWASP Foundation. https://owasp.org/www-project-internet-of-things/ (2023)
31. Popoola, S.I., Ande, R., Adebisi, B., Gui, G., Hammoudeh, M., Jogunola, O.: Federated Deep Learning for Zero-Day Botnet Attack Detection in IoT-Edge Devices. IEEE Internet of Things Journal **9**(5), 3930–3944 (2022). DOI 10.1109/JIOT.2021.3100755
32. RATS: Rough Auditing Tool for Security. https://security.web.cern.ch/recommendations/en/codetools/rat (2021)
33. RATS: CERN Computer Security Information. https://security.web.cern.ch/recommendations/en/codetools/rat (2023)
34. Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M.: Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In: International Conference on Machine Learning and Applications (ICMLA), pp. 757–762. IEEE, Orlando, FL (2018). DOI 10/ggssk7

35. Russell, R.L., Kim, L., Hamilton, L.H., Lazovich, T., Harer, J.A., Ozdemir, O., Ellingwood, P.M., McConley, M.W.: Automated Vulnerability Detection in Source Code Using Deep Representation Learning (2018). DOI 10.48550/arXiv.1807.04320
36. Somda, Y.: Guesslang - Detect the programming language of a source code (2021)
37. Swessi, D., Idoudi, H.: A Survey on Internet-of-Things Security: Threats and Emerging Countermeasures. Wireless Personal Communications **124**(2), 1557–1592 (2022). DOI 10.1007/s11277-021-09420-0
38. Viega, J., Bloch, J., Kohno, Y., McGraw, G.: ITS4: A static vulnerability scanner for C and C++ code. In: Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00), pp. 257–267 (2000). DOI 10.1109/ACSAC.2000.898880
39. Wang, H., Ye, G., Tang, Z., Tan, S.H., Huang, S., Fang, D., Feng, Y., Bian, L., Wang, Z.: Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. IEEE Transactions on Information Forensics and Security **16**, 1943–1958 (2021). DOI 10.1109/TIFS.2020.3044773
40. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In: International Conference on Neural Information Processing Systems (NeurIPS), p. 11. Curran Associates, Inc., Vancouver, Canada. (2018)