Empirical Software Engineering (2023) 28:38
https://doi.org/10.1007/s10664-022-10276-6

An empirical study of text-based machine learning models for vulnerability detection

Kollin Napier[1] · **Tanmay Bhowmik**[1] · **Shaowei Wang**[2]

Accepted: 13 December 2022 / Published online: 3 February 2023

Abstract

With an increase in complexity and severity, it is becoming harder to identify and mitigate vulnerabilities. Although traditional tools remain useful, machine learning models are being adopted to expand efforts. To help explore methods of vulnerability detection, we present an empirical study on the effectiveness of text-based machine learning models by utilizing 344 open-source projects, 2,182 vulnerabilities and 38 vulnerability types. With the availability of vulnerabilities being presented in forms such as code snippets, we construct a methodology based on extracted source code functions and create equal pairings. We conduct experiments using seven machine learning models, five natural language processing techniques and three data processing methods. First, we present results based on full context function pairings. Next, we introduce condensed functions and conduct a statistical analysis to determine if there is a significant difference between the models, techniques, or methods. Based on these results, we answer research questions regarding model prediction for testing within and across projects and vulnerability types. Our results show that condensed functions with fewer features may achieve greater prediction results when testing within rather than across. Overall, we conclude that text-based machine learning models are not effective in detecting vulnerabilities within or across projects and vulnerability types.

Keywords Vulnerability detection · Machine learning · Text-based analysis

Communicated by: Yuan Zhang

✉ Kollin Napier
kollin.napier@msstate.edu

Tanmay Bhowmik
tbhowmik@cse.msstate.edu

Shaowei Wang
shaowei.wang@umanitoba.ca

[1] Department of Computer Science and Engineering, Mississippi State University, Mississippi State, MS, USA

[2] Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada

1 Introduction

As of November 2022, over 600 websites have been affected by known security breaches resulting in nearly 12 billion unique accounts with leaked information.[1] Recent major security breaches affecting tens of millions of users have occurred at corporations such as Equifax, Home Depot, and Target (Plachkinova and Maurer 2018; Shu et al. 2017; Wang and Johnson 2018). It is estimated that at least 82% of Americans have had an account breached with many accounts being affected more than once (Cor and Sood 2018). Security breaches can have large financial impacts on businesses and consumers (Cavusoglu et al. 2004; Layton and Watters 2014; Telang and Wattal 2007) and come in various forms, most notably exploitation of vulnerabilities related to software systems.[2]

As software systems continue to be developed and maintained in various capacities, the associated source code can introduce a variety of vulnerabilities. As of November 2022, the Common Vulnerabilities and Exposures (CVE) database[3] has over 185,000 entries detailing vulnerabilities with more than 15,000 unique entries occurring in each of the last five years.[4] Compared to a decade ago, the number of vulnerabilities has doubled. These vulnerabilities can stem from a variety of reasons related to the source code including: reuse, implementation, dependencies, modifications, or developer error (Dowd et al. 2006; Pham et al. 2010; Piessens 2002). Additionally, online crowdsourced question and answer (Q&A) platforms, such as Stack Overflow, may also inadvertently introduce software vulnerabilities into development due to users copying and pasting answers in the form of code snippets (Abdalkareem et al. 2017; Fischer et al. 2017; Klock 2021; Zhang et al. 2021).

Software vulnerabilities can typically be avoided through code reviews before or after deployment by the initial developers or other developers within a team or across a community (Mäntylä and Lassenius 2008). However, it is possible that vulnerabilities survive with or without realization over a period (McQueen et al. 2009). Code reviews can be useful, but they can cost developers a lot of time and money exploring potential avenues for exploitation and determining appropriate solutions for mitigation. Given a large code base, it may also be impossible for each source code file to be inspected and fixed if associated with a vulnerability (Czerwonka et al. 2015).

To help minimize vulnerabilities, developers have adopted various methods such as source code analysis tools provided in a static or dynamic fashion (Li and Cui 2010). Static source code analysis is performed by examining the source code without executing any portion of a system. Dynamic source code analysis, on the other hand, requires execution of a system and numerous test cases involving various actions throughout a system, such as accepting user input (Egele et al. 2008). These tools can potentially help identify vulnerabilities within source code and reduce the overhead needed in determining a location, association, or mitigation of a vulnerability through code reviews.

Researchers have improved the methods of traditional analysis tools by introducing machine learning techniques to better identify software vulnerabilities (Ghaffarian and Shahriari 2017; Grieco et al. 2016; Ijaz et al. 2019; Jie et al. 2016; Lin et al. 2017; Spreitzenbarth et al. 2015; Yamaguchi et al. 2011). These methods utilize various machine learning

[1] https://haveibeenpwned.com
[2] https://owasp.org/www-project-top-ten/
[3] https://cve.mitre.org/cve/
[4] https://cvedetails.com/browse-by-date.php

algorithms in combination with features which can include source code or related software metrics to establish training data. However, such attempts typically focus on a single aspect such as software systems with one or more versions or single vulnerabilities, thus the detection method might only be applicable in one setting (Yamaguchi et al. 2011). In other attempts, various samples are used to establish training data, but the trained models are not tested across settings such as other systems (Chernis and Verma 2018; Harer et al. 2018; Shar et al. 2014).

In combination with machine learning, text-based analysis has also been introduced to help classify information such as vulnerabilities (Hovsepyan et al. 2012; Huang et al. 2010; Spanos et al. 2017; Wijayasekara et al. 2014). Data is typically acquired through text mining methods, which can provide a variety of information. Generally, a textual description is used (Spanos et al. 2017), however, it is possible to convert information such as source code to be interpreted as plain text (Hovsepyan et al. 2012).

Typically, the answers provided on Stack Overflow are posted as code snippets, meaning they are non-executable programs or do not contain the full context related to a code segment. A recent study related to the prevalence of security vulnerabilities via code snippets on the crowdsourced Q&A website Stack Overflow determined that 36% of the total Common Weakness Enumeration (CWE) vulnerability types were detected in over 600,000 C/C++ code snippets. Revisions to vulnerable code snippet answers were suggested in an attempt to mitigate vulnerabilities, but only half of their suggestions were adopted (Zhang et al. 2021). Although code snippets may come in a variety of forms, establishing a foundation of data representing such non-executable source code provides a structure in which the problem can be explored.

With a motivation to help combat the dominance of vulnerable code snippets, we elect to train and test text-based machine learning models by utilizing "fixed" and "vulnerable" extracted functions from various software systems. In this effort, we explore the usage of text-based machine learning models within and across projects and CWE vulnerability types. There is limited attention provided to the topic of cross-project and cross-type vulnerability detection using machine learning with the only examples being (Scandariato et al. 2014; Ban et al. 2019; Lin et al. 2019 and Liu et al. 2020). We have the assumption that predictors do not work well cross-project and cross-type due to the various characteristics of different types of vulnerabilities and systems (Li et al. 2017; Zou et al. 2019). Little is known about the effectiveness of text-based prediction of vulnerabilities using machine learning across projects or CWE vulnerability types.

With a goal to address this gap, in this paper, we conduct empirical studies to explore the accuracy of vulnerability detection using text-based analysis methods from extracted source code functions represented within and across various projects and CWE vulnerability types. Compared to related work, we utilize larger datasets composed of various software systems, avoid issues related to the class imbalance problem by creating equal function pairings between "fixed" and "vulnerable" samples, and perform extensive training and testing within and across various projects and CWE vulnerability types. We proceed by asking the following research questions:

- RQ1: Are text-based vulnerability detection techniques effective in detecting vulnerabilities across projects?
- RQ2: Are text-based vulnerability detection techniques effective in detecting vulnerabilities across vulnerability types?

To analyze the performance of text-based machine learning models, we use a combination of two datasets: 1) a 2.5 gigabyte (GB) database dump mapping CVE IDs to vulnerability contributing commits (VCCs) of GitHub projects (Perl et al. 2015), and 2) Big-Vul, a C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries (2002-2019) (Fan et al. 2020). Our initial findings suggest text-based analysis may provide slightly better vulnerability detection results when using reduced features (no context) from functions rather than whole (full context) functions. We performed an empirical analysis using over 99,000 function pairs extracted from 344 GitHub projects and around 80,000 function pairs from 38 Common Weakness Enumeration (CWE) vulnerability types corresponding to the projects. The pairs represent extracted C/C++ source code functions from available GitHub projects that have commits with an associated CVE ID. We map the available CVE related commits to CWE vulnerability types to produce additional testing data. We process a set of function pairings representing full functions and label the data as *Full Context* (FC). We also apply additional processing methods which focus on the differences between a pair of functions. The first method, called *No Context* (NC), focuses on differences based on whitespace, while the second method, called *Lines Context* (LC), looks into the line differences between the functions with available context. From our findings, a final analysis is performed using Convolutional Neural Network (CNN) machine learning models coupled with the Term Frequency-Inverse Document Frequency (TF-IDF) term-document matrix natural language processing technique on data processed with our *NC* method. We detect vulnerabilities within and across the top 10 projects and top 10 mapped CWE vulnerability types both based on the number of extracted pairs of functions. Our final analysis suggests that text-based vulnerability detection techniques are less effective in detecting vulnerabilities across projects and vulnerability types compared to detecting within.

The contributions of this paper lie in conducting an empirical examination on detecting vulnerabilities across projects and vulnerability types. We:

- Increase the scope of measurement by performing empirical statistical analysis.
- Determine if there is a statistically significant difference between within versus across prediction.
- Provide insight for possible reasoning behind the results for within versus across testing.

The rest of this paper is organized as follows: Section 2 discusses related work represented by heuristic-based and machine learning-based detection methods. Section 3 details our methodology including the dataset as well as data processing, function extraction and pairing. Section 4 describes the process for applying the data to machine learning models as well as our evaluation metrics. Section 5 details our preliminary experiment based on our initial processing method and introduces additional processing methods to determine the best combination of data, model and technique to use through results of random sampling. Section 6 describes and answers our research questions through additional analysis focused on specific projects and vulnerability types based on function pairings. Section 7 provides further discussion related to our overall results and conclusions. Section 8 details any threats to validity. Section 9 provides a final conclusion and description of future work.

2 Related Work

The concept of detecting vulnerabilities has been explored in various capacities and is continually adapted to suit the evolution of software development. Researchers are experimenting

with various methods and tools to improve vulnerability detection through training and testing of data and machine learning models. For detailing related work, we classify some existing static analysis vulnerability detection techniques into two categories: those which have not implemented machine learning and those which have.

Heuristic-Based Prior to the inclusion of machine learning algorithms, various tools and techniques have provided many options for software vulnerability detection. Li and Cui (2010) provide a detailed overview of static analysis tools and techniques by evaluating and comparing commonly used practices including lexical analysis, type inference, theorem proving, data flow analysis, model checking and symbolic execution. Through comparison of eight widely used tools, they conclude that no single tool is universal, and a combination of tools might provide a greater benefit.

Liu et al. (2012) also provide a survey related to the popular techniques such as static analysis, fuzzing, penetration testing and vulnerability discovery models (VDMs). They provide details for related work of each technique while comparing them by highlighting advantages and disadvantages of each. They conclude their survey by discussing the future development of each technique.

Shin and Williams (2008) utilize the disadvantage of complexity in software security to help predict problems by using metrics to prioritize inspection and testing. Using the metrics tool, Understand C++,[5] they collected nine complexity metrics. From the measurements, they compared vulnerable and non-vulnerable functions as well as vulnerable and fault-but-non-vulnerable functions. They conclude that vulnerable functions have distinctive characteristics from non-vulnerable and faulty-but-non-vulnerable functions. Their approach provides low false positive rates; however, it is possible that many vulnerabilities are still missed. They also state that their results might not compare to other systems and further research is needed regarding additional metrics.

Sultana et al. (2016) focus on using method level patterns to capture properties of a method, known as nano-patterns, to reduce security risks in software. Using known vulnerabilities in Apache Tomcat[6] they extract nano-patterns for methods used in fixing vulnerabilities. Through analysis of the nano-pattern distribution between vulnerable and non-vulnerable methods, they determined that certain nano-patterns are significantly present in vulnerable methods.

Machine Learning-Based Ghaffarian and Shahriari (2017) create an extensive review of various machine learning and data-mining techniques. From their review, they develop four main categories which summarized usage for relevant publications. These categories include: Vulnerability Prediction Models based on Software Metrics, Anomaly Detection Approaches, Vulnerable Code Pattern Recognition, and Miscellaneous Approaches. In relation to these categories, our work would fall under the Vulnerable Code Pattern Recognition. From conducting their survey, the authors conclude that several techniques have been successfully used in different application domains while highlighting any shortcomings. Overall, they state that the research involving vulnerability analysis with machine learning remains in an immature state.

Lin et al. (2020) present a survey related to usage of deep neural networks for identifying vulnerabilities as well as associated challenges. Researchers use a variety of networks

[5] https://scitools.com

[6] https://tomcat.apache.org/

for feature representation including: fully connected networks (FCNs), convolutional neural networks (CNNs), Recurrent neural networks (RNNs), among others. The work they reviewed is categorized into four types of feature representation: graph-based, sequence-based, text-based, and mixed. They present tables of reviewed studies which provide details including a reference to the paper, the data used, usage of labels, feature representations, and detection granularity. They also include discussion on challenges involving available data and model interpretability. They conclude that the state of vulnerability detection using neural networks is immature with several problems left to be solved.

Hovsepyan et al. (2012) present a novel approach for vulnerability prediction that analyzes raw source code as text for mobile applications on the Android platform. Using a Support Vector Machine for the training phase to build a prediction model, their initial results show high values for accuracy, precision, and recall, all averaging above 80%. However, their approach only uses a single Android application written in Java. Mokbal et al. (2019) and Mubarek and Adalı (2017) both incorporate neural-networks in their work and conclude that their proposed schemes outperform other traditional models.

Duan et al. (2019) propose a tool, VulSniper, to detect fine-grained vulnerabilities where it is otherwise hard to distinguish between vulnerable and non-vulnerable source code. Using a series of metrics, they compare their tool against benchmark datasets and achieve higher F1-scores. However, their experiment only focuses on two CWE vulnerability types. Their experimental results show an improved accuracy by utilizing an attention mechanism and they encourage future research to explore these new insights.

Wang et al. (2020) present a framework, FUNDED (Flow-sensitive vUlNerability coDE Detection), that uses graph neural networks (GNNs) to build vulnerability detection models. Their framework presents a method to capture and reason the control, data, and call dependencies within a program. They compare their framework by comparing it against seven other models at the function level. Their experiment results show their framework outperforms other approaches.

Li et al. (2021a) present a vulnerability detector, VulDeeLocator, aimed at providing high detection and high locating abilities. They provide solutions to common difficulties such as semantic relations and control flows by leveraging intermediate code and applying granularity refinement. Using arbitrary real-world software, they were able to detect 18 confirmed vulnerabilities with two not previously reported. However, their approach is limited to source code needing to be compiled. In addition, they state that more research is needed in the direction of explainability since they can only partly explain the effectiveness of their detector.

Li et al. (2021b) propose the first systematic framework to detect vulnerabilities in C/C++ source code using Bidirectional Recurrent Neural Networks (RNNs). Their framework, titled Syntax-based Semantics-based and Vector Representations (SySeVR), obtains representations related to vulnerabilities through syntax and semantic information by adapting a concept of region proposal. To evaluate the effectiveness of their framework, they utilize data from the National Vulnerability Database (NVD)[7] and Software Assurance Reference Database (SARD),[8] which creates a dataset containing 126 types of vulnerabilities. However, their approach is limited by using a single model and only trained to detect vulnerabilities in four software products.

[7] https://nvd.nist.gov
[8] https://samate.nist.gov/SARD/

Liu et al. (2019) develop a system, DeepBalance, to combine deep code representation and fuzzy-based class rebalancing to overcome challenges such as code representation and class imbalance. Using a Bidirectional Long Short-Term Memory (BiLSTM) model, they rebalance training data by generating synthetic samples for vulnerable code. They apply their system to several projects and show they can significantly improve the vulnerability detection when compared against other deep learning machine learning methods.

Lin et al. (2019) design a deep-learning framework using long-short term memory to extract relevant information from cross-domain datasets for vulnerability detection. Their framework handles varying scenarios where input data may or may not be labeled and include synthetic vulnerability samples from SARD to account for limitations of real world vulnerability data. They use abstract syntax trees (ASTs) to represent code syntax. However, they do acknowledge that their AST-level processing method may discard vulnerable code parts. From their experiments, they are able to determine the validity and effectiveness of their framework by deriving feasible and effective representations of vulnerable patterns. They conclude that their framework can outperform a baseline represented by the static vulnerability tool, Flawfinder.[9]

Perl et al. (2015) present a new method for detecting dangerous code within repositories by combining code-metric analysis with metadata from repositories. By training a Support Vector Machine model, they can flag suspicious commits and reduce a false alarm rate by 99% compared to another system, Flawfinder. Li et al. (2018) guide a deep learning-based vulnerability detection system, called Vulnerability Deep Pecker (VulDeePecker) by training it using representations of software programs known as code gadgets. They were able to achieve smaller false negative rates compared to other systems, but conclude their approach has several limitations including usage of a single BLSTM neural network.

Scandariato et al. (2014) provide a novel approach by text mining source code components and applying a bag-of-words method to detect vulnerabilities. Their study consisted of three experiments which utilized 20 Android applications and machine learning models Random Forest and Naive Bates. Their first two experiments determined that Random Forest achieved better performance in the majority of applications. Their third exploratory experiment determines if a model trained on one application can be used to detect vulnerabilities in another application. From their results, they conclude that some models built on a single application can predict vulnerable software components in other applications. However, they do not have any methods to determine which data can be used to improve the prediction models.

Ban et al. (2019) perform an evaluation study on deep-learnt features for software vulnerability detection using seven machine learning techniques including a Bidirectional Neural Network, Random Forest, and Support Vector Machine. They generate a dataset by collecting three open-source projects as well as the dataset provided by Li et al. (2018) involving CWE119 and CWE399 code gadgets. Their study is divided into three evaluations involving: single dataset with multiple vulnerabilities, cross-project vulnerability, and class imbalance for vulnerabilities. From their experiment results, they conclude that machine learning-based techniques suffer from mediocre performance for cross-project vulnerability detection due to a class imbalance problem. In comparison to their work, we utilize a larger dataset and create an equal number of "fixed" and "vulnerable" extracted source code functions pairings to avoid a data imbalance problem.

[9] https://dwheeler.com/flawfinder/

Based on existing literature, we provide an empirical analysis on the process of detecting vulnerabilities using text-based analysis methods within and across projects and CWE vulnerability types. We select the Random Forest (RF), Linear Support Vector Machine (LSVC), Multi-layer Perceptron (MLP), Bidirectional Long Short-Term Memory (BiLSTM), Convolutional Neural Network (CNN), Text Convolutional Neural Network (TextCNN), and Bidirectional Encoder Representations from Transformers (BERT) machine learning algorithms to gauge the best performance, given the used datasets and processing methods. In contrast to other works, our final analysis will focus on the metric of average precision score related to vulnerability detection across projects and CWE vulnerability types using the CNN algorithm and a set of processed data that avoids the class imbalance problem by using equal function pairings.

3 Methodology

In this section, we describe the methodology used in regards to the datasets and how we process specific data. Section 3.1 describes the datasets and their structure. Next, we mention how we collect data (Section 3.2) and proceed with processing it (Section 3.3). After processing, we describe how we extract functions from source code (Section 3.4). Finally, we state how functions are paired from a "fixed" and "vulnerable" version of source code (Section 3.5).

3.1 Datasets

This paper utilizes two datasets: 1) a 2.5 gigabyte (GB) database dump mapping CVE IDs to VCCs of GitHub projects originally contributed by Perl et al. (2015)[10] and 2) Big-Vul, a C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries (Fan et al. 2020).[11] The database dump consists of three tables: *commits*, *cves*, and *repositories*. Out of the 179 repositories, only 50 have associated records in the *commits* table and 42 out of those 50 have an associated *commits* record related to a *cve* record. Each record in the *commits* table has a *type* field which identifies the commit as "fixing_commit", "other_commit", or "blamed_commit".

A "fixing_commit" represents a commit which fixes a vulnerability with proof provided from a CVE entry and a commit message. Other commits that fixed a vulnerability but lacked more than one form of proof are considered as an "other_commit". A "blamed_commit" refers to a commit which introduced a vulnerability. Perl et al. (2015) defined this using a few heuristics involving ignoring changes in documentation, blaming deleted lines, blaming lines before and after and block of code. However, due to potential limitations in the determination of a "blamed_commit", we decide to focus on the "fixing_commit" and "other_commit" records. These records will be used to identify a parent commit which still contains a vulnerability. The following terms will be used throughout the paper in reference to commits, associated source code files, or functions:

– **"Fixed"**: Commit of type "fixing_commit" or "other_commit"; commit that mitigated a vulnerability

[10] The original database link provided by the paper is unavailable, but an alternative link was found: https://github.com/announce/vcc-base

[11] https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset

– **"Vulnerable"**: Parent commit of "Fixed"; last instance of a vulnerability

The C/C++ code vulnerability dataset curated by Fan et al. (2020), named Big-Vul, provides a variety of files, but we only focus on the initial file which contains CVE entries covering a period of 2002 to 2019 consisting of 21 features. We utilize the following features to reduce the data down to necessary information: *project*, *cve_id*, *version_after_fix*, and *version_before_fix*.

In both datasets, we are identifying a commit which "fixed" a vulnerability and a commit which contains a "vulnerability". In this case, the parent commit is identified as still being vulnerable because it contains an instance of source code files prior to a defined fix. In doing so, we do not claim the parent commit introduced the vulnerability. To further investigate the validity of this assumption, we randomly selected 100 commits across various projects and found that 90% of files modified within the commits are associated with fixing vulnerabilities. This can be further assessed by viewing the CVE entry as it pertains to the identifier associated with the commit of a given project. However, we do acknowledge that not all modified functions correspond to mitigation.

3.2 Data Collection

For the database dump, only projects which have an associated *commits* and *cves* record are considered, for a total of 42 usable projects and over 72,000 commits from the original 351,409 records in the *commits* table. Each project, if available, is cloned from GitHub to a local machine for processing. Using the version control history of git, we retrieve source code files from projects and set their content based on previous modifications. This process is achieved by using their SHA-1 hash value from the *commits* table.

The Big-Vul dataset follows a similar approach to the database dump. Only available open-source projects on GitHub which have a CVE ID and contain data related to "fixed" and "vulnerable" versions are considered. However, in this case, the "vulnerable" version is provided for us, so we do not have to retrieve it. Before processing and removing duplicates in comparison to the database dump, we have a total of 365 projects with over 3,700 records. Once the data from both datasets has been collected, we then process accordingly to determine the overall amount of "fixed" and "vulnerable" functions. Figure 1 displays an overview of the entire process, with the following subsections providing the details.

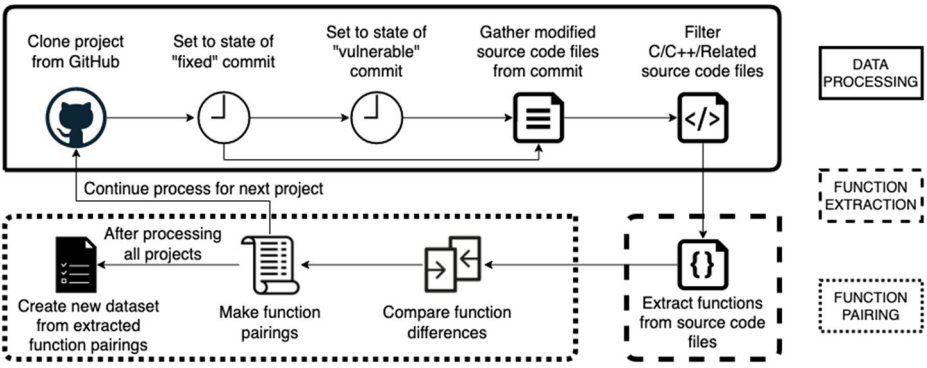


Fig. 1 Overview of our methodology for creating dataset

3.3 Data Processing

To process appropriate data from the cloned projects, we retrieve a parent ("vulnerable") commit for each "fixed" commit to find the code differences between the changes. Only existing files which have been modified between the two commits are used, so any file which is created or deleted is discarded. Only desired programming language file extensions and filenames are considered which helps initially filter out unnecessary files. In this case, those which correspond to a programming language and do not contain rejected keywords, such as "test", are considered. For this study, only C/C++ are considered for the remaining portion due to C related files being the overwhelming majority across all projects in the database dump and C++ being close enough in structure related to C. Additionally, the Big-Vul dataset was already created to only contain C/C++ data.

3.4 Function Extraction

We provide a lightweight approach to parse source code files for available functions based on a series of regular expressions and conditions. From our motivation based on incomplete/uncompilable code snippets, we elect to construct our own methods. We present an overview of our extraction process in Algorithm 1. The function extraction process begins by looking for a function declaration. We highlight a regular expression used for identifying C/C++ function declarations in Listing 1. A C/C++ function declaration can be defined using various specifiers and keywords such as *public*, *static*, *void*, among other options. An example of a function is shown in Listing 2. Other regular expressions are used to identify pieces such as function body, conditions, loops, and end. Additional conditions are defined to offer support in identifying syntax based on what was previously observed, such as parentheses or brackets. These were constructed from a manual process in which we verified valid identification of functions in a sample of source code files. As a result, this approach provides a focused dataset by mitigating potential noise that could be irrelevant to the overall analysis.

The function example in Listing 2 represents a publicly accessible function that prints out the string "Hello World" and does not return any value. Although this could be a typical format for functions, not all follow this scheme. There might be a variety of declarations due to coding formats and preferences from developers or configurations.

The overhead for processing individual files will vary based on the length and complexity. For example, functions containing several nested conditions may take longer to process than those with general conditions that only contain a body.

This process is continued for all available functions within a source code file. In the end, all available functions across source code files from commits of all projects are combined into a set of "fixed" and "vulnerable" functions, creating a balanced set of data. This helps avoid the class imbalance problem where the amount of "fixed" samples might outweigh the amount of "vulnerable" samples.

3.5 Function Pairing

After the C/C++ related source code files have been parsed and available functions extracted, each function is compared to its counterpart version to determine if there is an actual difference between them. A difference in this case can be a single character change or

```
Data: Fixed and vulnerable source code files from projects
Result: Identify and extract functions from source code data
initialization;
Regular expressions for patterns such as function start, function end, body, loops, and
conditions
for s in source code files do
    Determine available files (source code from projects)
    for c in file contents do
        Read the contents of each source code file, line by line
        if comments or comment block then
            Ignore comments, comment blocks or similar and continue to next line of
            source code files
        else
            Otherwise, read current line and identify pattern
            if found function then
                - Keep track of found function (declaration), leading spaces, brackets,
                parentheses, and other characters in determining body, loops,
                conditions, and other nested data
                - Set flags as needed to determine current point in process
                - Define series of conditions to determine if the current line is the
                continuation or end of a function
            else
                Continue
            end
        end
    end
end
```

Algorithm 1 Extract functions from source code.

multiple lines within the function. If a difference is not found between the versions, then the functions are considered invalid for pairing and discarded. This helps minimize any confusion for the machine learning models during the training phase. For example, if the contents of a "fixed" function and "vulnerable" function are the same, then the model may not be able to properly identify a function later when testing. Listings 3 and 4 provide examples of a "fixed" and "vulnerable" version of a function.

Once functions have been compared to determine if a difference is present, they can begin to be paired. The pairs will be split into two sets of data, one with all "fixed" functions and the other with all the associated "vulnerable" functions. Creating pairs allows for an equal number of functions, minimizing any problem due to imbalanced data. After all files are processed, a total of over 199,000 functions are extracted to create over 99,000 pairings.

```
(static|bool|char|int|float|double|void|wchar_t|struct).*\(.*\)
    *.*\{*
```

Listing 1 Regular expression to identify C/C++ function declarations

```
public static void() {
/* This prints out Hello World */
printf(``Hello World'');
}
```

Listing 2 Example of a function

4 Data Preparation for Machine Learning Models

4.1 Model Data Processing

Once a balanced set of data has been processed from extracted "fixed" and "vulnerable" functions, it can be used to train machine learning models. Hereafter, we refer to the data from this processing method as *Full Context* (FC) since it represents a function in its full (regular) state. When processing the data, each function will be assigned a value of 0 ("vulnerable") or 1 ("fixed") for the model to comprehend. As the entry of the functions in the overall pairs are done on a project-by-project basis, meaning the first 100 functions could relate to a single project, a process of random sampling needs to occur. Such random sampling can ensure that the data tested can come from various projects and not be biased towards an individual project. To test the effectiveness of a model, we utilize the *train_test_split* class from *scikit-learn* to split the data into 75% training and 25% testing subsets for the models to examine and test accuracy.

Following the work of Scandariato et al. (2014), Li et al. (2018), Kim et al. (2019), Liu et al. (2020), and Tang et al. (2020), we combine our functions into a text corpus and apply separate natural language processing (NLP) techniques including term document and bag-of-words methods Count and Term Frequency-Inverse Document Frequency (TF-IDF) as well as embedding-based methods Word2Vec (Mikolov et al. 2013a; 2013b),[12] Doc2Vec (Le and Mikolov 2014),[13] and Bidirectional Encoder Representations from Transformers (BERT) preprocessing/encoding (for the BERT pre-trained model) (Devlin et al. 2018) to create different text representations of the source code for comparison. Utilizing the Python libraries scikit-learn,[14] gensim,[15] Keras,[16] and TensorFlow,[17] we tokenize the different text corpus of functions and use them to train and test our machine learning models. Table 1 provides an example of how a single "fixed" function from the corpus can be processed with different NLP techniques.

4.2 Machine Learning Models

By implementing, training, and testing seven machine learning models, we are able to test the average precision of vulnerability detection for the *Full Context* (FC) data based on our five NLP techniques: Count, TF-IDF, Word2Vec, Doc2Vec, and BERT. The machine

[12] https://radimrehurek.com/gensim/models/word2vec.html
[13] https://radimrehurek.com/gensim/models/doc2vec.html
[14] https://scikit-learn.org/
[15] https://radimrehurek.com/gensim/
[16] https://keras.io
[17] https://tensorflow.org

```
void xics_set_irq_type(XICSState *icp, int irq, bool lsi) {
assert(ics_valid_irq(icp->ics, irq));
icp->ics->islsi[irq - icp->ics->offset] = lsi;
}
```

Listing 3 Example of "fixed" function from data

learning models include Random Forest (RF), Multi-layer Perceptron (MLP), Linear Support Vector Classification (LSVC), Bidirectional Long Short-Term Memory (BiLSTM), Convolutional Neural Network (CNN), Text Convolutional Neural Network (TextCNN), and Bidirectional Encoder Representations from Transformers (BERT). These models are implemented via the Python libraries scikit-learn, Keras and TensorFlow.

The implementation of these models is done from a baseline perspective, meaning we elect to use default parameters as they are defined in the manuscripts or documentation from the libraries in which we generate the models from. For example, with RF we use 100 estimators (trees in the forest). For the MLP we have a default value of 100 layers which creates a hidden layer size of 98, *n_layers - 2* with a stochastic gradient-based optimizer named "*adam*". For LSVC we use a loss function of "*squared_hinged*" which corresponds to the square of the hinge loss (standard SVM loss). Following the works of Li et al. (2018) and Ban et al. (2019), we implement the BiLSTM using layers of bidirectional 64 LSTM units for a total of 128. Although CNN is mentioned in several papers we referenced, there was not an active implementation, so we elect to implement our own with 128 filters (number of output filters in the convolution) and a "*relu*" activation function. The TextCNN model is based on the work by Kim (2014) and Chen (2015) which utilizes convolutional layers with multiple filter widths. The BERT model is based on a pre-trained design of deep bidirectional representations from unlabeled data (Devlin et al. 2018). This data is derived from the BooksCorpus by Zhu et al. (2015) and English Wikipedia[18] which combine for over 3 billion words. We elect to include BERT in our analysis as it can provide better performance in natural language processing tasks compared to other models (Qiu et al. 2020; Koroteev 2021). Further modifications to the model configurations could be performed to make adjustments as needed.

Usage of the BERT model differs from the other six models in that it is pre-trained, meaning it has already been created and trained on data to solve a similar problem. Implementation of this model through TensorFlow requires utilization of existing BERT preprocessing and encoding for our corpus and labels. The pre-trained model is version 4 provided by the TensorFlow Hub[19] which is uncased and uses 12 hidden layers, a hidden size of 768, and 12 attention heads. Throughout the remainder of this study, data corresponding to BERT will be provided in an alternative manner such as a stand-alone plot or analysis, given its uniqueness.

4.3 Evaluation Metric

To analyze the performance of our models, we focus on the average precision (AP) scoring metric for each model. From the Related Work detailed in Section 2, a precision or recall metric is primarily used (Ban et al. 2019; Liu et al. 2020; Scandariato et al. 2014). In this

[18] https://wikipedia.org

[19] https://tfhub.dev/tensorflow/bert_en_uncased_L-12_H-768_A-12/4

```
void xics_set_irq_type(XICSState *icp, int irq, bool lsi) {
ICSState *ics = icp->ics;
assert(ics_valid_irq(ics, irq));
ics_set_irq_type(ics, irq - ics->offset, lsi);
}
```

Listing 4 Example of "vulnerable" function from data

case, precision measures the ability of the model to not label a negative ("vulnerable") sample as positive ("fixed"). Whereas a recall measurement shows the ability of the model to find all positive ("fixed") samples. The AP score provides a combination of these metrics by calculating an average of the prediction scores provided by the trained models. It summarizes a precision-recall curve as the weighted mean of the precision (P) achieved at each threshold with the increase in recall (R) from the previous threshold used as the weight. The equation of AP can be represented as:

$$AP = \sum_{n} (R_n - R_{n-1}) P_n$$

where P_n and R_n present the precision and recall at the nth threshold (Su et al. 2015; Zhu 2004).

Since we are performing a classification problem, we provide additional metrics using box plots for *Full Context* (FC) data during the Preliminary Experiment. These metrics include: accuracy, precision, recall, and the area under a receiver operating characteristic curve (ROC AUC) as shown in Fig. 3. The scores are a combination of results from all models and NLP techniques used. For the remainder of the study, due to similar trends,

Table 1 Example output from applying NLP techniques to a function

Step	Result
Source Code	`void xics_set_irq_type(XICSState *icp, int irq, bool lsi){assert(ics_valid_irq(icp->ics, irq));icp->ics->islsi[irq - icp->ics->offset] = lsi;}`
Vocabulary	`{'void': 10, 'xics_set_irq_type': 11, 'xicsstate': 12, 'icp': 2, 'int': 5, 'irq': 6, 'bool': 1, 'lsi': 8, 'assert': 0, 'ics_valid_irq': 4, 'ics': 3, 'islsi': 7, 'offset': 9}`
Feature Names	`['assert', 'bool', 'icp', 'ics', 'ics_valid_irq', 'int', 'irq', 'islsi', 'lsi', 'offset', 'void', 'xics_set_irq_type', 'xicsstate']`
Term-Document Matrix	`[[1 1 4 3 1 1 3 1 2 1 1 1 1]]`
TF-IDF Matrix	`[[0.14586499 0.14586499 0.58345997 0.43759497 0.14586499, ...]]`
Word Vector Matrix	`[[ 9.7702928e-03, 8.1651136e-03, 1.2809718e-03, 5.0975787e-03, 1.4081288e-03, ...]]`
Document Vector Matrix	`[[-3.8822503e-03 3.4151897e-03 8.9458562e-04 -7.8290445e-04, -4.7795074e-03, ...]`

additional box plots from further experiments are presented in the Appendix to minimize redundancy.

5 Preliminary Experiment

In this section, we describe the process for our preliminary experiment regarding the seven machine learning models and sets of data from our five NLP techniques. First, we describe our environment setup used for the preliminary experiments as well as additional experiments throughout the study (Section 5.1). Then, we describe our process of using five iterations of randomly sampled data from all projects to be used for testing the prediction of our models (Section 5.2). Next, we discuss the results from the *Full Context* (FC) data processing method (Section 5.3). From these results, we introduce our additional data processing methods, *No Context* (NC) and *Lines Context* (LC) (Section 5.4). Next, we describe the results from these additional data processing methods (Section 5.5) and further analyze all the methods using five random samples of 10,000 pairs (Section 5.6). Finally, we perform Kruskal-Wallis and Bonferroni tests to determine the best machine learning model, NLP technique, and data processing method to use going forward in our study (Section 5.7).

5.1 Environment Setup

In performing our preliminary experiment, we elect to use the available hardware from an Apple iMac system. This iMac has a 3.4 gigahertz (GHz) quad-core Intel Core i5 processor. This machine also contains 32 gigabytes (GB) of 2400 megahertz (MHz) of double data rate 4 synchronous dynamic random-access memory (DDR4 SDRAM). We continue to use this machine throughout the remaining experiments in our study.

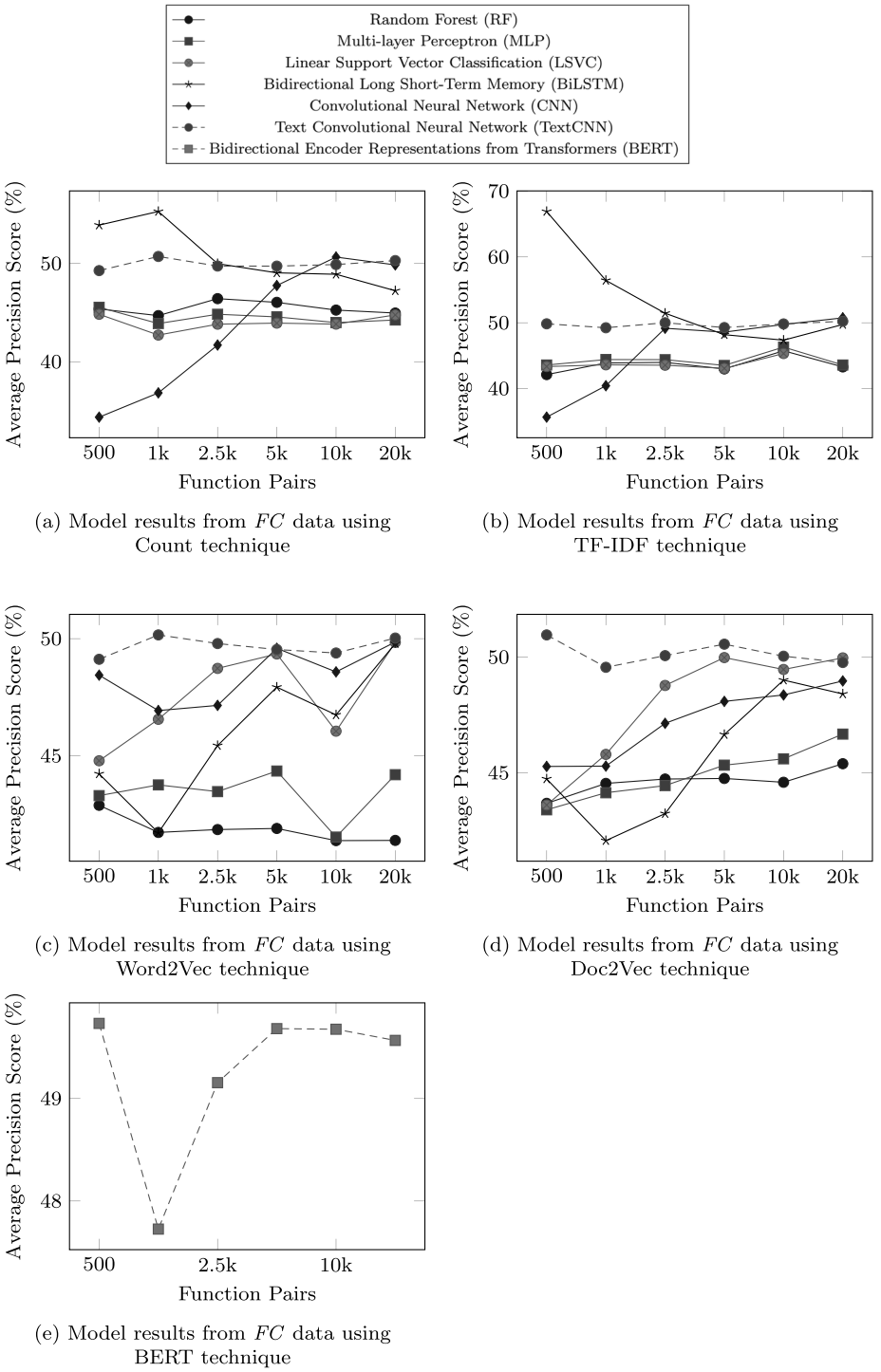
5.2 Data Sampling

The data used in the preliminary experiment as well as other experiments in our study are publicly available in a GitHub repository.[20] A series of random sample sizes, which include functions from a portfolio of projects, are selected to determine the average precision score (APS) of the vulnerability detection rate. Sample sizes include: 500, 1,000, 2,500, 5,000, 10,000, and 20,000 random function pairs. We choose a maximum of 20,000 random function pairs due to required processing time and power for training each model. In conjunction with our random samplings, we elect to repeat the process five times to generate a mean APS given the fluctuation of the data available in the random samples. The same random sample of *FC* data will be given to each of the NLP techniques for each iteration. After the data is processed accordingly, it will be split and used to train and test the models.

5.3 Results from *FC* Data

The results in Fig. 2 show a mean APS across all trained and tested models using each sample size and NLP technique for the *Full Context* (FC) data. BERT is presented in a separate plot Fig. 2e since the other NLP techniques cannot be applied to the BERT model nor can the BERT preprocessing and encoding be applied to the other models. Overall there is a

[20] https://github.com/krn65/emse_data



(a) Model results from *FC* data using Count technique

(b) Model results from *FC* data using TF-IDF technique

(c) Model results from *FC* data using Word2Vec technique

(d) Model results from *FC* data using Doc2Vec technique

(e) Model results from *FC* data using BERT technique

Fig. 2 Mean APS across all models using *FC* random samples for all NLP techniques

similar trend among the techniques, where Count and TF-IDF appear to stay within a consistent range, while Word2Vec, Doc2Vec, and BERT increase over time with some dips. For each of the models we observe different trends based on the NLP techniques being used. For example, the TextCNN model scoring drops with an increase in samples for the Count and TF-IDF techniques, but increases with the Word2Vec and Doc2Vec techniques. Whereas, the CNN model maintains an upward trend throughout all of the techniques. Similar to other models, BERT levels off around 5,000 samples, hovering around 50%. Although all models appear to increase or be consistent overtime, the RF and MLP models degrade with use of the Word2Vec techniques and increasing samples.

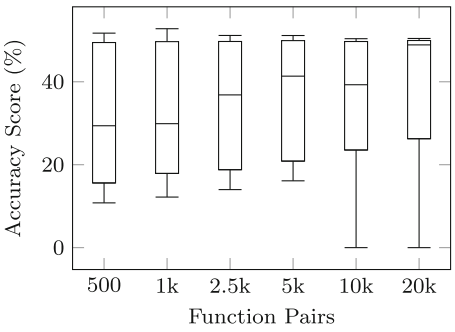
Increasing from the initial 500 random samples, we notice drops in some cases likely due to an adequate increase in the number of features. Given the small sample size, we continue looking beyond this point to develop a better understanding of how the models perform. After 5,000 samples, all models appear to maintain a consistent range with the top models in all cases hovering around the 50% average precision mark.

From the five NLP techniques we utilize,Word2Vec appears to achieve the highest mean values across the different models and sample sizes when using *FC* data. Doc2Vec offers the second highest with an upward trend occurring after 2,500 samples. Additionally, the TF-IDF, Count, and BERT techniques follow, hovering between 40% and 50%. Across the series of samples, the distribution of models that perform well against each other can be dependent upon the NLP technique and amount of data. While each NLP technique achieves at least one instance where the APS reaches near or above 50%, it appears that none of the models perform well given most results fall below 50%. This means the prediction from each model is slightly worse than what we consider a "best guess" attempt at classifying a function as "fixed" or "vulnerable".

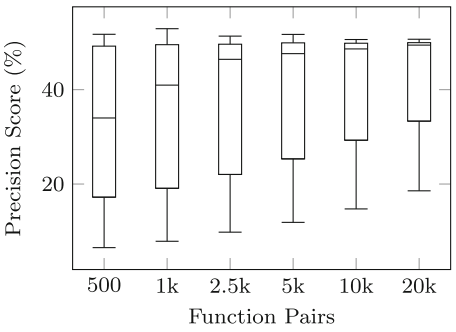
The results from Fig. 3 provide an overview of additional metrics for a combination of all models and NLP techniques using the *FC* data. For the series of random sample function pairings for each metric, we observe a wide range of scores with an increase in pairs over time. Accuracy represents the scoring of labels that were exactly predicted. With an increase in samples and overall features, the accuracy suffers over time. Precision is defined by the ratio of True Positives / (True Positives + False Positives) which represents the ability for the models to not mislabel data (i.e., labeling a positive sample as negative). In contrast to accuracy, the precision improves with an increase of data. Recall is defined as the ratio of True Positives / (True Positives + False Negatives) which represents the ability of the models to find all positive samples. From these results, the model may achieve high recall, but the majority of results hover around 50%. The ROC AUC score is defined by the performance of a binary model from thresholds in which the curve data is summarized as a single number. As with the accuracy and precision, the results fall within a range of 40%.

5.4 Additional Processing Methods

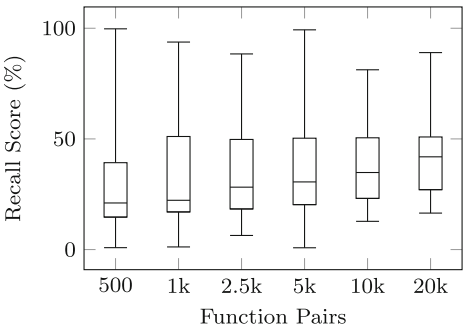
Since the model results, using data from our *Full Context* (FC) processing method, are below a "best guess" attempt, we further test our data by reducing the overall features. Although only functions which had a difference between the "fixed" and "vulnerable" versions were initially considered when creating a function pairing, there is still a possibility of excess features which can cause confusion for the models during the training phase. For example, it could be possible for a 1% difference to occur between the function versions but the remaining 99% is the same. To avoid this issue, we introduce two new data processing methods which reduce both the "fixed" and "vulnerable" versions of the functions down to differences between them while pairing remains intact.



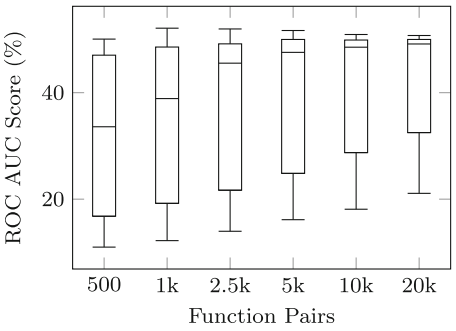
(a) Accuracy results from all models and NLP techniques using *FC* random samples in Preliminary Experiment



(b) Precision results from all models and NLP techniques using *FC* random samples in Preliminary Experiment



(c) Recall results from all models and NLP techniques using *FC* random samples in Preliminary Experiment



(d) ROC AUC results from all models and NLP techniques using *FC* random samples in Preliminary Experiment

Fig. 3 Additional metrics from all models and NLP techniques using *FC* random samples in Preliminary Experiment

For the first data processing method, each pair of functions are split into the portions based on whitespace where each piece is compared against the counterpart function to determine if that piece exists in both. If so, that piece is discarded. Otherwise, it is kept as a difference between the two. The result presents a list of unique features that occur between the "fixed" and "vulnerable" versions. We label this additional method as *No Context* (NC) since we are removing the majority of context within a function.

The second data processing method is similar to the *NC* method, but instead of splitting a function into portions based on whitespace, we use semicolons to represent lines of code. We compare each line of a function against its counterpart function to determine a difference. However, this time we focus on the lines (context) of a given function to create a code snippet. Following the work of Li et al. (2018), we utilize a guiding principle for vulnerability detection where a line of code containing a vulnerability may depend on its context. The idea behind this method refers to the assumption that a line of code may rely on various other lines within a source code file. For this instance, we are assuming that lines which

occur before and after a specific line may help provide additional context. With the resulting differences, we find the occurrence of that line from the full function and proceed to extract two lines that occurred before and two lines that occurred after if they are available and not already present. We minimize our context using two lines with the assumption that a higher number of lines might create confusion for a model with increased features as seen with data from the *Full Context* (FC) processing method. We label this data processing method as *Lines Context* (LC).

Table 2 displays a function pairing ("fixed" and "vulnerable" function) from our data and how it is handled using the *Full Context* (FC), *No Context* (NC), and *Lines Context* (LC) data processing methods. In this example, *Full Context* refers to the function in its raw state with comments removed. The *NC* row displays the output of data that is present in one version, but not the other. Similarly, the *LC* row displays a line difference with added context.

Table 3 provides a deeper look at how an extracted function is processed by the *LC* method. In this example, the "vulnerable" function is shown in various steps throughout the process. First, the function appears in its original state with comments removed. Next, the function is split into pieces based on semicolons which represent lines of source code. Then, we determine the differences by comparing each piece to the counterpart function ("fixed"). Finally, the leftover differences are used to determine if, from their location in the original split list, there is any available context that does not already exist in the differences list. If so, then append that context to our leftover differences. Although this processing method can help in adding context, it may also re-introduce similarities between the two versions of a function. Also, it is possible for a function to be processed using *LC*, but still appearing as it did in its original form, especially for functions with small length.

After the entire data is processed using the additional methods, the models are again trained and tested. In training and testing models using the *NC* and *LC* methods, we take an alternative approach to test models given a "real world" scenario. For example, from our ability to process data using these additional methods, we rely on the pairing of functions that have been created during the extraction process. After pairing, we process these reduced functions with fewer features. In the instance we are evaluating a snippet of code for a vulnerability, we may not have the ("fixed" or "vulnerable") counterpart available to process the function. Therefore, when testing our models we need to utilize the *Full Context* (FC) data to create an applicable result.

For this process, we initialize a splitting procedure for training and testing data. Previously, when training and testing on the *Full Context* (FC) data, we were able to apply a class which splits the data into 75% training and 25% testing for the overall corpus. However, we are now working with two sets of data. To keep our evaluation of sample sizes inline, we manually take 75% of the current sample size for the *No Context* (NC) or *Lines Context* (LC) and 25% of the current sample size of the *FC* data for our training and testing. For example, if we have 500 samples, then we will have 375 (*NC*/*LC*) samples for training and 175 (*FC*) samples for testing. As before, we utilize five iterations of random sample sizes to generate a mean APS due to fluctuation. This process helps us evaluate any difference in prediction results given the variation of data processing methods, machine learning models, and NLP techniques.

5.5 Results from Additional Methods

From the results in Figs. 4 and 5, we observe the usage of condensed functions with less features may offer an advantage over full functions with more features. Overall, the *LC* data

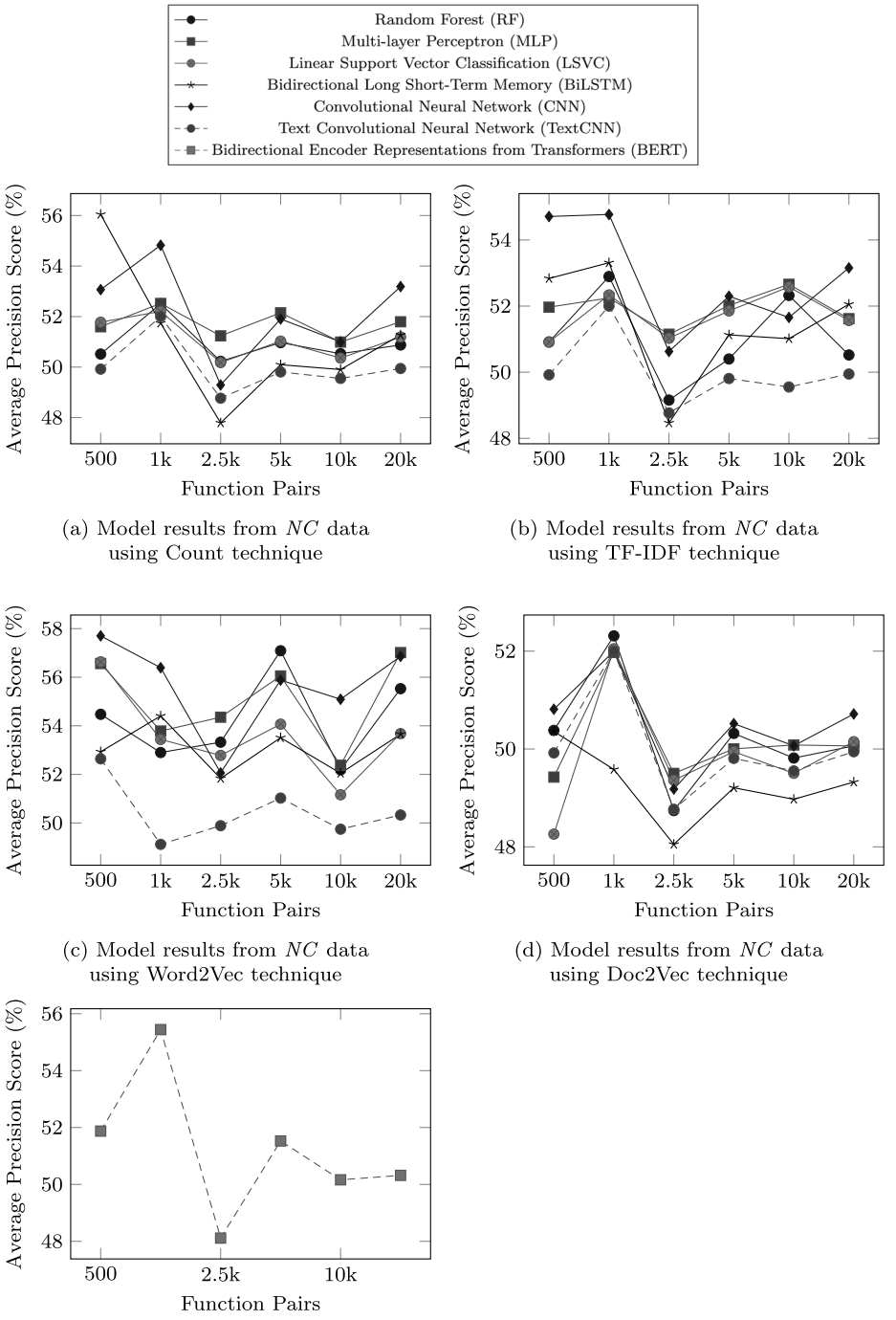
Table 2 Example of data processing methods handling function differences

Processing Method	Type (Length)	# Features	"Fixed" Function
FC	String (1)	10	`static int raw_truncate(BlockDriverState * bs, int64_t offset){ return bdrv_truncate( bs->file, offset); }`
NC	List (4)	4	`['bdrv_truncate(bs->file,', 'offset);']`
LC	List (1)	10	`['static int raw_truncate(BlockDriverState *bs, int64_t offset){ return bdrv_truncate( bs->file, offset)']`
Processing Method	**Type (Length)**	**# Features**	**"Vulnerable" Function**
FC	String (1)	17	`static int raw_truncate(BlockDriverState * bs, int64_t offset){ BDRVRawState *s = bs-> opaque; if (s->type != FTYPE_FILE)return -ENOTSUP; if (ftruncate(s->fd, offset)< 0) return -errno; return 0; }`
NC	List (18)	12	`['BDRVRawState', '*s', '=', 'bs->opaque;', 'if', '(s->type', '!=', 'FTYPE_FILE)', '-ENOTSUP;', 'if', '(ftruncate(s->fd,', 'offset)', '<', '0)', 'return', '-errno;', 'return', '0;']`
LC	List (4)	17	`['static int raw_truncate(BlockDriverState *bs, int64_t offset){ BDRVRawState *s = bs->opaque', 'if (s->type != FTYPE_FILE )return -ENOTSUP', 'if (ftruncate(s->fd, offset)< 0)return -errno', 'return 0']`

Table 3 Example of generating LC data

Steps	"Vulnerable" Function
Original	`static uint32_t lan9118_readl(void *opaque, target_phys_addr_t offset){ lan9118_state *s = (lan9118_state *)opaque; if (offset < 0x20){ return rx_fifo_pop(s); } switch (offset){ case CSR_GPT_CNT: return ptimer_get_count(s->timer); case CSR_WORD_SWAP: return s->word_swap; case CSR_FREE_RUN: return (qemu_get_clock_ns(vm_clock)/ 40)- s->free_timer_start; } hw_error ("lan9118_read: Bad reg 0x%xn", (int)offset); return 0; }`
Split	`['static uint32_t lan9118_readl(void *opaque, target_phys_addr_t offset){ lan9118_state *s = (lan9118_state *)opaque', 'if (offset < 0x20){ return rx_fifo_pop(s)', '} switch (offset){ case CSR_GPT_CNT: return ptimer_get_count (s->timer)', 'case CSR_WORD_SWAP: return s->word_swap', 'case CSR_FREE_RUN: return (qemu_get_clock_ns(vm_clock)/ 40)- s-> free_timer_start', '} hw_error("lan9118_read: Bad reg 0x%x\\n", (int)offset)', 'return 0', ' }']`
Differences	`['case CSR_FREE_RUN: return (qemu_get_clock_ns (vm_clock)/ 40)- s->free_timer_start']`
Without Context	`case CSR_FREE_RUN: return (qemu_get_clock_ns(vm_clock)/ 40)- s-> free_timer_start }switch (offset){ case CSR_GPT_CNT: return ptimer_get_count(s->timer); case CSR_WORD_SWAP : return s->word_swap; case CSR_FREE_RUN: return (qemu_get_clock_ns(vm_clock)/ 40)- s-> free_timer_start;; } hw_error("lan9118_read: Bad reg 0x%x\n", (int)offset); return 0;`
With Context	

shows little improvement over the *FC* data, while the *NC* data exceeds 50% average precision in the majority of scenarios. We also note that the CNN model appears to outperform the other models in most cases. Compared to the other data processing methods, *NC* yields a higher mean APS across all scenarios. In both the *NC* and *LC* data, we observe a fluctuation of each model given an increase in random samples. Regardless of technique used, a pattern of highs and lows is created throughout the process. In comparison to the *FC* data, we see varying trends. For example, when using *LC* data, we observe an initial spike at 1,000 random samples then a sharp decrease followed by a seemingly steady score going forward. The BERT model appears to slightly benefit more from the *NC* data compared to the *FC* and *LC*. Although, the usage of *FC* data appears consistent whereas the *LC* produces a decreasing trend until after 10,000 random samples. We assume that results may be dependent on the data being used. For example, some features may hold a greater weight compared to



(a) Model results from *NC* data using Count technique

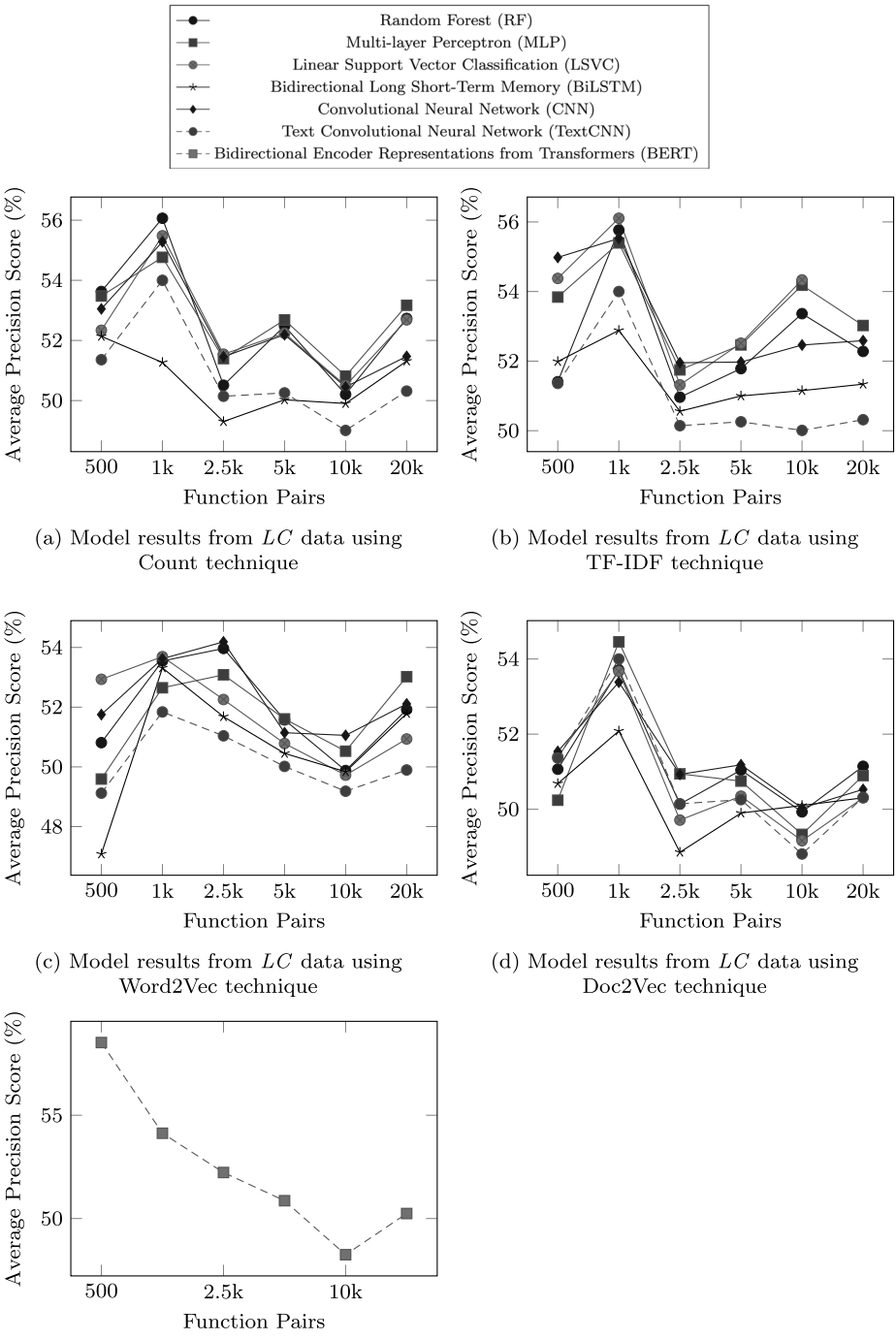
(b) Model results from *NC* data using TF-IDF technique

(c) Model results from *NC* data using Word2Vec technique

(d) Model results from *NC* data using Doc2Vec technique

(e) Model results from *NC* data using BERT technique

Fig. 4 Mean APS across all models using *NC* random samples for all NLP techniques



(a) Model results from *LC* data using Count technique

(b) Model results from *LC* data using TF-IDF technique

(c) Model results from *LC* data using Word2Vec technique

(d) Model results from *LC* data using Doc2Vec technique

(e) Model results from *LC* data using BERT technique

Fig. 5 Mean APS across all models using *LC* random samples for all NLP techniques

others from the random samples. From these results, we elect to perform additional analysis to determine, based on statistical evidence, which machine learning model, NLP technique, and data processing method we should use going forward for additional experiments.

5.6 Further Analysis of Models' Performance

We want to further investigate our findings and determine if the machine learning models, natural language processing (NLP) techniques, or our data processing methods have a significant influence on the average precision score (APS). To that end, we formulate the following null and alternative hypotheses:

On Machine Learning Models:

H_{10}: There is no significant difference between the machine learning models when training and testing on a random sample of function pairings from various projects.

H_{11}: There is a significant difference between the machine learning models when training and testing on a random sample of function pairings from various projects.

On NLP Techniques:

H_{20}: There is no significant difference between the NLP techniques when training and testing on a random sample of function pairings from various projects.

H_{21}: There is a significant difference between the NLP techniques when training and testing on a random sample of function pairings from various projects.

On Data Processing Methods:

H_{30}: There is no significant difference between the data processing methods when training and testing on a random sample of function pairings from various projects.

H_{31}: There is a significant difference between the data processing methods when training and testing on a random sample of function pairings from various projects.

We notice that the data we are dealing with here is not normally distributed. Therefore, in order to test our hypotheses, we conduct Kruskal-Wallis (Kruskal and Wallis 1952) tests, a non-parametric statistical ranked-based test. Our objective is to determine if there is a difference between the groups which represent the average precision scores of the machine learning models, NLP techniques, and our data processing methods. For these experiments, we choose a maximum of 10,000 pairs since going beyond this amount appears to offer little to no advantage in most cases as shown in Figs. 2, 4, and 5.

5.7 Kruskal-Wallis Tests

To execute our Kruskal-Wallis tests, we define a set of independent and dependent variables for each test. First, we set our models as the independent variable and their related average precision scores as the dependent variable. Next, we set our NLP techniques as the independent variable and their related average precision scores as the dependent variable. Finally, we set our data processing methods as the independent variable and their related average precision scores as the dependent variable. Additionally, we perform a Bonferroni correction (Dunn 1961) test to mitigate any Type-1 errors which provide false positive readings from our Kruskal-Wallis results.

Since we are utilizing multiple models, NLP techniques, and data processing methods, we generate data points for all possible combinations (N = 60). However, since BERT is a pre-trained model, we could not apply data using other NLP techniques to the model nor could we apply data pre-processed and encoded with BERT to the other models. Therefore, the amount of data points for BERT (N = 15) is less than the other combinations. Given

a difference in the amount of data points, we elect to define two Kruskal-Wallis tests for the machine learning models: 1) between the first six models (RF, MLP, LSVC, BiLSTM, CNN and TextCNN) and 2) between the highest mean ranking combination (model, NLP technique, and data processing method) and BERT.

Table 4 provides a mean ranking of the machine learning models which shows the CNN model achieving the highest ranking, followed by BiLSTM with the remaining MLP, LSVC, RF, and TextCNN models. Table 5 suggests that the models do not have a significant difference between them based on their mean APS at $\alpha = 0.05$ level of significance with $p = 0.068$. Table 6 shows a mean ranking of the natural language processing (NLP) techniques with TF-IDF having the highest mean ranking, followed by the Word2Vec, Count and Doc2Vec techniques. However, Table 7 suggests that the NLP techniques do have a significant difference between them based on their mean APS at $\alpha = 0.05$ level of significance with $p = < 0.001$. Finally, Table 8 provides a mean ranking for our data processing methods which shows the *NC* method achieving the highest mean ranking. Table 9 suggests that there is a significant difference in our data processing techniques based on their mean APS at $\alpha = 0.05$ level of significance with $p = < 0.001$.

Based on the aforementioned statistical evidence, we fail to reject our null hypothesis (H_{10}) for the machine learning models. For the NLP techniques, we reject our null hypothesis, (H_{20}), and accept our alternative hypothesis, (H_{21}). Finally, we reject our null hypothesis (H_{30}) and accept our alternative hypothesis (H_{31}) for our data processing methods.

Tables 10, 11, and 12 provide further results through an estimates comparison for the machine learning models, NLP techniques, and data processing methods. According to Table 10, CNN (as shown by the Kruskal-Wallis Test) may generate a better mean than the other models. For the NLP techniques, Table 11 shows TF-IDF with the highest mean rank. Additionally, the *No Context* (NC) data processing method achieves a higher mean rank than the *Lines Context* (LC) or *Full Context* (FC) methods.

Based on the mean rankings from our initial Kruskal-Wallis tests, we determine that a combination of the Convolutional Neural Network (CNN) model, TF-IDF technique, and a *NC* data processing method appear to be the best approach. With this combination, we now have an equal amount of data points (N = 15) to make a comparison with remaining BERT model. Tables 13 and 15 show that our determined combination achieves a greater mean value and rank than BERT. Additionally, Table 14 suggests there is a significant difference between these two models based on their mean APS at $\alpha = 0.05$ level of significance with $p = 0.007$. Following this analysis, we could conclude that the *NC* data processing method coupled with the Convolutional Neural Network (CNN) model and the TF-IDF technique may achieve the best results. For the remainder of this study, we elect to use this combination for further experiments related to our research questions (Table 15).

Table 4 Kruskal-Wallis test - ranks - models

Model	N	Mean Rank
Random Forest (RF)	60	168.95
Multi-layer Perceptron (MLP)	60	178.92
Linear Support Vector Classification (LSVC)	60	175.67
Bidirectional Long Short-Term Memory (BiLSTM)	60	179.88
Convolutional Neural Network (CNN)	60	217.35
Text Convolutional Neural Network (TextCNN)	60	162.33
Total	360	

Table 5 Kruskal-Wallis test - test statistics - models

	AP
Kruskal-Wallis H	10.256
df	5
Asymp. Sig.	0.068

Table 6 Kruskal-Wallis test - ranks - NLP techniques

Model	N	Mean Rank
Count (BOW)	90	169.38
TF-IDF (BOW)	90	220.10
Word2Vec	90	182.70
Doc2Vec	90	149.82
Total	360	

Table 7 Kruskal-Wallis test - test statistics - NLP techniques

	AP
Kruskal-Wallis H	21.921
df	3
Asymp. Sig.	< 0.001

Table 8 Kruskal-Wallis test - ranks - data processing methods

Model	N	Mean Rank
Full Context (FC)	120	91.60
No Context (NC)	120	232.37
Lines Context (LC)	120	217.53
Total	360	

Table 9 Kruskal-Wallis test - test statistics - data processing methods

	AP
Kruskal-Wallis H	132.588
df	2
Asymp. Sig.	< 0.001

Table 10 Bonferroni test on models

Model	Mean	Group
Convolutional Neural Network (CNN)	50.763	A
Bidirectional Long Short-Term Memory (BiLSTM)	49.578	B
Text Convolutional Neural Network (TextCNN)	49.546	B
Linear Support Vector Classification (LSVC)	49.332	B
Multi-layer Perceptron (MLP)	49.032	B
Random Forest (RF)	48.766	C

Table 11 Bonferroni test on NLP techniques

NLP Technique	Mean	Group
TF-IDF (BOW)	50.53	A
Word2Vec	49.249	B
Count (BOW)	49.208	B
Doc2Vec	49.023	B

6 Answering the Research Questions

6.1 (RQ1) Are text-based vulnerability detection techniques effective in detecting vulnerabilities across projects?

Given the excessive number of features for our *Full Context* (FC) data containing full extracted function pairs across various projects, our preliminary experiment was only able to utilize a random sample of data. To help narrow down our focus and address our first research question, we explore projects on an individual basis.

As shown in Table 16, the top 10 projects, based on function pairings, will be chosen for training and testing within and across each other. This will help identify if one model has a comparable detection ability across projects versus being trained and tested on itself.

For this analysis, we will use a model validation technique known as cross-validation. This technique will divide the data into separate folds, which are used as a testing set. For this study, 10 stratified folds will be used on the data for a given project for training and testing within itself. Given a set of data, each fold will contain an equal percentage of samples. We proceed with utilizing the data from the *No Context* (NC) processing method along with the TF-IDF technique and Convolutional Neural Network (CNN) model to train and test each project.

6.2 Inside Cross-Project Vulnerability Detection Results

The results related to the trained and tested models for within and across the top 10 projects are found in Table 17. This table provides an average precision score percentage for detecting vulnerable code in relation to the trained data which is labeled as fixed (1) or vulnerable (0) for machine learning input. From our previous statistical analysis, we elected to use the *NC* data, TF-IDF technique, and CNN model. As with the preliminary experiment using *NC* data, we train on *NC* data and test on *FC* data to better represent a "real world" scenario.

The highest average precision score for each row (project) is bold. When observing the table we see that the bold values occur in a diagonal, representing the highest scores that frequently occur when training and testing a model on itself using 10-fold cross-validation.

Training and testing within a project yields a range of 52.07% to 63.36% with an average of 57.17%. Training and testing across a projects yields an average range of 49.85% to

Table 12 Bonferroni test on data processing methods

Data Processing Method	Mean	Group
No Context (NC)	50.94	A
Lines Context (LC)	50.58	A
Full Context (FC)	46.982	B

Table 13 Kruskal-Wallis test - ranks - models (Additional)

Model	N	Mean Rank
Convolutional Neural Network (CNN)	15	19.87
Bidirectional Encoder Representations from Transformers (BERT)	15	11.13
Total	30	

50.71% with the average being 50.25%. The lowest within score occurs at P4 (*tovalds/linux*) with 52.07% while the highest occurs at P6 (*ellson/graphviz*) with 63.36%. The lowest score when cross-testing occurs between P5 (*chromium/chromium*) and P10 (*inspircd/inspircd*) with 48.16%. Meanwhile, the highest score when cross-testing occurs between P5 and P7 (*jktjkt/trojita*) at 53.09%. Compared to our preliminary experiments, the results are relatively similar. If we observe the scores for testing across other projects, we notice a fluctuation in several areas where there are high and low points of scoring, with some only showing a slightly better score than the "best guess" attempt of 50%. Overall, the average of the cross-testing is roughly 50%.

Overall, it appears that the top three projects, which had significantly more function pairs and features compared to the other projects, did not perform as well as lower paired and featured projects when testing across or even within. It can be assumed there is likely a lot of "noise" left within the data or features that provide little to no weight regarding their value towards the overall corpus per the labeled "fixed" and "vulnerable" data. This may also suggest that there is still a possibility for confusion when a model is testing on full functions ("real world" data). Additionally, either a case of too much data or not enough data could also play a role. To further evaluate the average precision scores within and across projects, we formulate the following null and alternative hypotheses:

H_0: There is no significant difference between the training and testing methods for project models when testing within itself versus testing across other projects.

H_1: There is a significant difference between the training and testing methods for project models when testing within itself versus testing across other projects.

Using the within and average of the across percentages from Table 17, we conduct a Kruskal-Wallis non-parametric statistical ranked-based test to determine if there is a significant difference between the testing methods for trained projects. Table 18 provides a ranking among the testing methods based on the mean average precision score. The within testing methods show a higher rank versus the across testing method from the trained machine learning models based on the top 10 projects from function pairings. Table 19 provides the results of our Kruskal-Wallis test and suggests that there is a significant difference among the methods used for testing within vs across at $\alpha = 0.05$ level of significance with $p = < 0.001$. From these results, we reject our null hypothesis (H_0) and accept our

Table 14 Kruskal-Wallis test - test statistics - models (Additional)

	AP
Kruskal-Wallis H	7.381
df	1
Asymp. Sig.	0.007

Table 15 Bonferroni test on models (Additional)

Model	Mean	Group
Convolutional Neural Network (CNN)	51.28	A
Bidirectional Encoder Representations from Transformers (BERT)	49.36	B

alternative hypothesis (H_1). In turn, we now answer RQ1 by stating that text-based vulnerability detection techniques are not effective in detecting vulnerabilities across projects.

Although our previous tests highlight the influence data can have on the results, the average precision score related to the random sample of function pairs across a variety of projects is still sub-par. Going back to the datasets, we decide to utilize a key factor in which they were established for, by using the associated CVE IDs related to the commit data. This provides motivation for our RQ2 in which we utilize the same methodology and shift our focus from a wide variety of data into smaller instances, such as a related CWE vulnerability type. Through a process of mapping associated CVE IDs from commits to a CWE vulnerability type, we process a new set of data using the *NC* method which focuses on source code related to individual CWE vulnerability types across various projects.

6.3 (RQ2) Are text-based vulnerability detection techniques effective in detecting vulnerabilities across vulnerability types?

To answer our second research question, we introduce an alternative approach in our vulnerability detection methods by utilizing a key piece of the datasets, a Common Vulnerabilities and Exposures (CVE) identifier (ID). A CVE ID is used to identify a unique vulnerability record within the CVE database. A record provides publicly available data detailing a vulnerability, usually with a description and references to acknowledgement as well as mitigation. A CVE ID can be mapped to a Common Weakness Enumeration (CWE) type. CWE provides a catalog for weaknesses and vulnerabilities related to software.

The process of mapping a CVE ID to a CWE type could be automated or manual, however a database called CVE Details[21] already exists which provides additional detail for a given CVE ID, including a related CWE type. Using the CVE Details database, the unique CVE IDs from the usable commits can be mapped to the CWE types.[22]

Using the relevant records retrieved from the *commits* table of the database dump, a total of 1,064 unique CVE IDs are present throughout all projects. Although this is a large amount, not all unique CVE IDs would provide enough data for the models to learn from. In this case, our mapping of CVE IDs to CWE types reduces the overall total to 40 available CWE types.

Table 20 provides data related to the top CWE vulnerability types sorted by the amount of function pairings. Additional information including the amount of unique CVE IDs from the overall data which correspond to those types as well as the amount of features from the *NC* data processing method for the available function pairings for the top 10 is provided. Given instances of undefined or unavailable data, we label them as such. From the original 4,111 unique CVE IDs, only 2,182 are now usable as they can be mapped to a CWE type. In total, 38 unique CWE vulnerability types are used to complete our mapping to functions related

[21] https://cvedetails.com

[22] CVE Details does provide a disclaimer that the site and all data are provided "as is", meaning it is not guaranteed to be accurate or complete.

Table 16 Top 10 projects based on number of function pairs

#	Project	Function Pairings	*NC* Features
1	ffmpeg/ffmpeg	40,828	35,207
2	bonzini/qemu	30,002	34,345
3	xen-project/xen	22,266	27,537
4	torvalds/linux	2,080	5,518
5	chromium/chromium	1,789	5,584
6	ellson/graphviz	427	1,335
7	jktjkt/trojita	199	643
8	libvirt/libvirt	197	531
9	adaptivecomputing/torque	166	566
10	inspircd/inspircd	134	746
	Total	98,088	112,012

to appropriate commits across all projects. The table only focuses on CWE vulnerability types which identified at least 1 applicable function pairings. We identified 69 other CWE vulnerability types which are not presented due to lack of usable data. A snippet of those not presented include: CWE 79 (47 unique CVE IDs), 284 (38), 415 (31), among others.

Based on the results we already analyzed from the experiments and tests involving the projects, we continue with a Covolutional Neural Network (CNN) model coupled with a new iteration of our *NC* data focused on functions relevant to our CWE mapping. As before with the projects, we take the top 10 CWE types based on function pairs to be trained and tested within itself and across other types. Cross-validation using 10 folds will be used to divide data for a CWE type and trained and tested within itself.

6.4 Inside Cross-Type Vulnerability Detection Results

Table 21 shows the results for the training and testing within and across CWE types. Based on our previous statistical analysis, we elected to use the *NC* data, TF-IDF technique, and CNN model. As with the preliminary experiment using *NC* data, we train on *NC* data and test on *FC* data.

Table 17 Average precision scores within and across top 10 projects using *NC* data with TF-IDF technique and CNN model

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
P1	**53.02**%	50.21%	50.37%	49.73%	51.07%	49.77%	49.72%	50.83%	51.33%	49.79%
P2	50.46%	**57.93**%	50.26%	50.58%	50.17%	49.36%	48.52%	51.28%	51.87%	49.21%
P3	50.24%	49.92%	**54.65**%	50.39%	50.44%	50.52%	49.62%	51.04%	51.89%	52.32%
P4	50.18%	50.19%	50.16%	**52.07**%	50.37%	49.48%	49.88%	50.01%	50.20%	48.28%
P5	50.25%	50.01%	50.38%	50.76%	**62.06**%	50.91%	53.09%	50.06%	51.16%	48.16%
P6	50.16%	50.38%	50.13%	49.71%	50.36%	**63.36**%	49.74%	50.10%	49.23%	48.84%
P7	50.28%	50.22%	50.26%	50.06%	50.15%	50.04%	**62.59**%	50.57%	51.08%	48.54%
P8	50.31%	50.25%	50.31%	50.38%	50.23%	50.43%	51.78%	**53.53**%	50.56%	48.92%
P9	50.13%	49.91%	50.09%	50.45%	50.48%	51.21%	49.80%	50.22%	**56.25**%	50.24%
P10	50.15%	49.96%	50.26%	49.85%	50.30%	49.62%	51.49%	50.11%	50.91%	**56.28**%

Table 18 Kruskal-Wallis test - ranks - project testing methods

	Method	N	Mean Rank
AP	Within	10	15.50
	Across	10	5.50
	Total	20	

CWE vulnerability types which were trained and tested within using 10-fold cross-validation achieve an average score of 57.86%. The lowest of these scores is C1 (*CWE 119*) with 55.20%. Meanwhile, the highest within score occurs at C7 (*94*) with 61.46%. Overall, the majority of cross test results all achieve above 50% with the lowest occurring between C10 (*CWE 17*) and C8 (*CWE 125*) with 49.61%.

Compared to the projects, testing across CWE vulnerability types offers a similar result hovering around 50%, displaying a "best guess" attempt. Although the CWE approach was more curated, the data still stems from various projects whereas the project approach only uses files available in a single project. While an increase in function pairings or features may seem beneficial, this is not the case as the top three CWE types do not offer the best results compared to the remaining seven. We further analyze these results as we did with the projects and formulate the following null and alternative hypotheses:

H_0: There is no significant difference between the training and testing methods for CWE type models when testing within itself versus testing across other CWE types.

H_1: There is a significant difference between the training and testing methods for CWE type models when testing within itself versus testing across other CWE types.

Table 22 provides a ranking among the methods based on the mean. The within testing method shows a higher rank versus the across testing method. Table 23 provides the results of our Kruskal-Wallis test and suggests that there is a significant difference among the methods used for testing within vs across at $\alpha = 0.05$ level of significance with $p =< 0.001$. From these results, we reject our null hypothesis (H_0) and accept our alternative hypothesis (H_1). In turn, we now answer RQ2 by stating that text-based vulnerability detection techniques are not effective in detecting vulnerabilities across vulnerability types.

7 Discussion

From our experiments involving various projects and CWE vulnerability types, we develop a better understanding of how machine learning models coupled with several types of processed data and natural language processing (NLP) techniques may affect the results regarding vulnerability detection. By identifying, extracting, and pairing "fixed" and "vulnerable" functions from source code, we use our text-based approach to identify an applicable machine learning model, NLP technique, and data processing method based on

Table 19 Kruskal-Wallis test - statistics - project testing methods

	AP
Kruskal-Wallis H	14.286
df	1
Asymp. Sig.	< 0.001

Table 20 Number of function pairings for CWE types mapped to unique CVE IDs

CWE type	# of CVE IDs	Function Pairings	*NC* Features
119	591	33,738	50,122
20	404	12,342	26,823
264	192	8,375	18,406
189	126	6,431	13,870
399	225	5,763	14,344
200	238	2,863	9,099
94	18	2,119	6,384
125	335	1,577	3,705
120	25	1,561	5,380
17	28	1,197	3,716
16	6	978	
190	109	870	
787	164	540	
310	36	323	
59	18	203	
134	4	203	
362	110	90	
400	44	85	
369	30	69	
476	164	66	
89	4	43	
193	8	34	
416	188	13	
269	22	11	
909	10	10	
255	3	9	
835	26	8	
352	2	5	
401	14	5	
772	34	5	
191	9	4	
287	19	4	
326	4	4	
665	7	4	
682	2	2	
129	6	1	
611	2	1	
754	4	1	
...			
Unavailable	5		
Undefined	475		
Total	4,111	79,557	
Total Usable	2,182	75,966	151,849

Table 21 Average precision scores within and across top 10 CWE types using *NC* data with TF-IDF technique and CNN model

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
C1	**55.54**%	51.10%	52.13%	50.95%	51.60%	51.77%	52.48%	50.16%	51.46%	50.02%
C2	50.97%	**58.00**%	51.38%	50.31%	50.46%	50.59%	53.05%	49.84%	51.02%	51.15%
C3	50.92%	50.92%	**58.45**%	50.50%	51.25%	50.87%	50.06%	49.73%	50.51%	50.82%
C4	50.55%	50.38%	50.45%	**55.19**%	50.99%	50.45%	50.19%	50.56%	50.39%	50.18%
C5	50.57%	50.59%	50.86%	50.99%	**57.42**%	50.56%	50.16%	50.20%	50.31%	50.63%
C6	50.59%	50.38%	50.40%	50.25%	50.54%	**58.61**%	51.34%	50.40%	50.93%	50.04%
C7	50.59%	50.74%	50.58%	50.25%	50.34%	50.99%	**61.61**%	49.90%	49.99%	50.07%
C8	50.30%	50.17%	50.37%	50.50%	50.32%	50.25%	50.15%	**56.02**%	50.71%	50.33%
C9	50.30%	50.41%	50.50%	50.77%	50.66%	50.15%	49.83%	50.28%	**58.23**%	50.55%
C10	50.17%	50.12%	50.45%	50.14%	50.14%	50.08%	49.90%	50.38%	49.69%	**58.59**%

the results from our experiments and statistical tests. We determine that our *Full Context* (FC) data processing method representing full functions does not provide the best results compared to our additional methods (*NC* and *LC*) representing condensed functions. We assume this is due to the number of available features causing confusion for the models. Specifically, the higher similarity between functions, the harder it may be for the models to distinguish the differences between a "fixed" and "vulnerable" function. For the remainder of this section, we provide further insights regarding results from our empirical study.

7.1 Within vs. Across

We can posit that the average precision scores can be higher within (i.e., projects and CVE vulnerability types) rather than across potentially due to several factors. One factor we mentioned previously was feature weight. In this context, weight represents the importance of features as they pertain to their syntax. It is possible for the features related to a particular project or CWE vulnerability type to vary in weight. For example, features from a particular project may weigh heavily in terms of usage while the usage of the same features is limited or non-existent in another. Thus, the approach of training and testing on differences from one might not be applicable for the other. We examine the similar occurrence of features between the *NC* training data and the *FC* testing data for the projects and CWE vulnerability types in Tables 24 and 25.

Features of Projects Table 24 presents the occurrence of similar features between the *No Context* (NC) training data and the *Full Context* (FC) testing data. For example, we observe the similar *NC* training features from P1 (*ffmpeg/ffmpeg*) that occur in testing features from P2 (*bonzini/qemu*) as 3,780 from the total 34,345 features between the *NC* and *FC* data

Table 22 Kruskal-Wallis test - ranks - CWE testing methods

	Method	N	Mean Rank
AP	Within	10	15.50
	Across	10	5.50
	Total	20	

Table 23 Kruskal-Wallis test - statistics - CWE testing methods

	AP
Kruskal-Wallis H	14.286
df	1
Asymp. Sig.	< 0.001

that P2 has. Continuing across the other projects tested using the trained model from P1, we notice a decrease in the amount of similar features. We see that the number of similar features tends to decrease in conjunction with the number of function pairings and overall features for the projects as shown in Table 16. However, from the results in Table 17, this does not appear to effect the average precision scores from the trained P1 model. The prediction between P1/P2 is 50.21%, meanwhile the prediction between P1/P9 (*adaptive-computing/torque*) is 51.33%, even though there are only 806 similar features between them. Regardless of the trained model, we see that an abundance of features makes little to no difference in the prediction result compared to those with lesser features when testing across. Similarly, we observe the within testing for the projects and see that the highest value 63.36% occurs in P6 (*ellson/graphviz*) with only 1,335 features compared to the 35,207 present in P1, which only had a 53.02% prediction. So, even though a decrease is present, this does not prove that the lack of similar features is a primary cause. Thus, as we assume, the weight of certain features could provide better results.

Features of Vulnerabilities Applying the aforementioned principle to vulnerabilities, the variety of potential vulnerabilities that may exist in one project or vulnerability type can fluctuate in another. We assume this makes it harder for a model to identify vulnerabilities which may not share characteristics in terms of features. Table 25 provides the occurrence of similar features between the *NC* training data and the *FC* testing data. In comparison to the projects, the overall occurrence of similar features across the vulnerabilities tends to be higher. However, this can also be attributed to a more focused set of data which created a higher number of function pairings per vulnerability type for the top 10 that were trained and tested.

Table 24 Similar feature occurrence between *NC* and *FC* across top 10 projects

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
P1	**35,207**	3,780	3,271	2,336	1,668	836	503	688	806	732
P2	3,646	**34,345**	6,126	3,143	1,482	765	492	862	891	722
P3	2,924	4,939	**27,537**	3,518	1,311	671	473	800	879	663
P4	1,029	1,325	1,585	**5,518**	548	349	233	387	407	345
P5	812	808	745	614	**5,584**	351	317	354	353	351
P6	382	356	325	264	261	**1,335**	115	142	161	124
P7	241	250	243	220	241	126	**643**	139	150	154
P8	200	230	225	199	162	115	84	**531**	135	116
P9	258	267	261	221	186	138	106	151	**566**	154
P10	358	364	354	312	289	176	173	194	227	**746**

Table 25 Similar feature occurrence between *NC* and *FC* across top 10 CWE vulnerability types

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
C1	**50,122**	32,095	19,460	23,844	25,443	14,361	15,109	10,331	13,972	6,433
C2	22,169	**26,823**	15,036	11,695	15,570	10,772	8,986	4,856	8,439	6,718
C3	14,818	13,916	**18,406**	6,121	11,262	9,415	5,428	1,545	5,213	6,296
C4	12,246	9,820	4,665	**13,870**	8,986	4,105	6,227	5,470	6,066	1,583
C5	12,396	10,855	7,856	7,629	**14,344**	5789	5218	3,910	5,198	4,015
C6	7,472	6,857	6,591	4,143	5,618	**9,099**	3,942	1,224	3,745	3,106
C7	5,803	5,081	3,768	4,321	3,988	3,628	**6,384**	1,766	3,482	1,139
C8	3,461	2,913	851	3027	2760	773	1,893	**3,705**	2,003	561
C9	4,869	4,241	3,095	3,668	3,598	2,888	3,026	1642	**5,380**	1,006
C10	3,095	3,217	3,140	1,103	2,760	2,225	947	619	948	**3,716**

The highest similar feature occurrence of two CWE vulnerability types occurs between C1 (*CWE 119*) and C2 (*CWE 20*) with 32,095. However, this intersection does not offer the highest average precision score for the C1 row of results when testing across (cf. Table 21). Like the projects, a lesser similarity produces the highest across-testing prediction with C1 and C7 (*CWE 94*) at 52.48% and 15,109 similar features. From the results in Tables 24 and 25, there is no conclusive evidence that the number of function pairings or features correspond to a higher average precision score when testing across. These observations suggest that further investigation of specific features across projects and vulnerability types is needed to provide a better conclusion.

7.2 Role of Project Structure and Application Domain

When focusing on projects, an assumption can be made that the form of its structure impacts the features, which in turn impacts the results. For example, with the source code that makes up a project, the way it is implemented or how the developer chooses to style the source code could play a role in how a model interprets specific features, such as variables within a function. It is also possible for projects to be composed of several programming languages and associated files. Thus, a vulnerability could be present within a project, but not within the specific files that are being examined. In our case, the projects supplied from the datasets are primarily developed using C/C++. However, it is possible for other aspects of the project to branch beyond this language.

Each of the projects are also unique in their own aspect given their designed purpose and overall functionality. For example, the project *ffmpeg/ffmpeg* provides a collection of tools to process media.[23] Whereas the project *bonzini/qemu* is defined as an emulator and virtualizer.[24] There is limited similarity in the features as shown in Table 24 and an underlying contrast between the functionality of a project could be a major difference. Further exploration on this assumption is needed where a new dataset can be used which comprises various projects that fall within a specific application domain, such as web browsers.

[23] https://github.com/FFmpeg/FFmpeg
[24] https://github.com/bonzini/qemu

7.3 Role of CWE Vulnerability Types

Although a vulnerability is a weakness within a system that can be exploited, not all are conceived in the same manner or exploit the same things. For example, CWE 119 is classified as "Improper Restriction of Operations within the Bounds of a Memory Buffer"[25] which deals with memory location. Whereas, CWE 20 is classified as "Improper Input Validation"[26] which corresponds to a system not properly validating input. Although these two types could potentially overlap, their overall classification and definition is different. However, this difference might not impact vulnerability types in the same way it might for projects. In comparison to the data corresponding to the top 10 projects, a focused dataset may increase detection across as shown by the data in Tables 20, 21, and 25.

7.4 Presence of CWE Vulnerability Types in Projects

Given the projects being utilized contain vulnerability data, and we are observing CWE vulnerability types, we further investigate the data by determining if the presence of specific vulnerabilities dominant specific projects. By using the vulnerability data corresponding to the mapped CVE IDs to the CWE vulnerability types, we further map the CWE vulnerability types to specific projects and determine their occurrence based on the number of unique CVE IDs present. Table 26 presents this data for the top 10 projects and top 10 CWE vulnerability types that were previously used. From this data we observe the presence of the majority of the top 10 CWE vulnerability types within the top 5 projects. The projects P4 (*torvalds/linux*) and P5 (*chromium/chromium*) appear to contain the most out of any likely due to their overall size and structure. For the remaining five projects (P6-P10), we see little or no presence of CWE vulnerability types (and CVE IDs). Other CWE IDs which do not appear in the top 10 have presence within the projects, most notably the larger projects such as P4 and P5. Some CWE IDs not mentioned include: CWE 476, CWE 787, CWE 362, among others.

We do note that some CWE vulnerability types do appear to be "dominant" within a specific project. Overall, we observe three cases for the projects. In one case we observe a single dominant vulnerability type given the amount of unique CVE IDs present, such as P1 and C1 (*CWE 119*). In a second case, we observe more than one "dominant" type as seen in projects P4 and P5. In this specific case we see at least two CWE vulnerability types which are dominant with C1 and C6 (*CWE 200*) for P4 and C1 and C2 (*CWE 20*) in P5. In the third case, we observe little to no occurrence.

It can be assumed that the lack of vulnerability presence in the remaining five projects would affect the average precision scores we observed, however, this is not the case. Referring to Table 17, the difference of the amount of CVE IDs does not appear to be an influence in the prediction regardless of the trained model. Although, if a project does not have enough vulnerabilities to be trained on, then testing across other projects for vulnerability prediction cannot be sufficient. For example, a model cannot be expected to test for other vulnerabilities and make a prediction, when there were no vulnerabilities for the model to be trained on. Overall, the usage of text-based machine learning models for vulnerability detection appears to strongly depend on the dataset and usage of another dataset may not be comparable. Along with this, other factors likely play a role for which we aim to explore in future work.

[25] https://cwe.mitre.org/data/definitions/119.html
[26] https://cwe.mitre.org/data/definitions/20.html

Table 26 Presence of top 10 CWE vulnerability types in top 10 projects based on CVE IDs

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
P1	61	2	0	15	6	1	1	10	2	0
P2	25	5	3	6	2	2	3	1	0	
P3	11	15	16	1	8	4	0	0	0	0
P4	139	107	82	43	84	130	1	23	9	12
P5	133	147	46	23	86	57	7	40	6	8
P6	3	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	1	0	0	0	0
P8	3	1	7	4	0	0	0	0	0	
P9	1	0	0	0	0	0	0	0	0	0
P10	1	2	1	1	0	0	0	0	0	

8 Threats to Validity

In this paper, we have presented results related to an empirical study of text-based machine learning models for vulnerability detection across projects and CWE vulnerability types. Using a combination of datasets, coupled with machine learning models, natural language processing techniques, and our own data processing methods, we conducted various experiments and statistical tests to achieve our findings. However, our work does have limitations which can affect the validity of this research.

8.1 Construct Validity

Construct validity relates to determining if the measurement used is an accurate representation of a variable (Bates and Cozby 2017). Throughout our paper, we focus on the metric of average precision score which summarizes a precision-recall curve as the weighted mean of precision scores from predictions (Su et al. 2015; Zhu 2004). It is possible for other metrics to be utilized which may offer alternative scores for any of the tests performed. However, we believe that regardless of the metrics used, the overall findings of our work would still hold.

8.2 Internal Validity

Internal validity is defined from the accuracy of the conclusions drawn (Bates and Cozby 2017). Due to the rich nature of the provided data and described methods, it is possible for our data processing methods and results to be exposed to some internal validity threats. For instance, the first dataset used, the database dump, came from another source link than that provided in the source paper. It is possible that the data could be incomplete, however, there is no indication of missing data during the process of parsing the database records. Additionally, the authors of the database dump did perform a manual analysis of 15% of the their mappings in which they determined an error rate of 3.1% (Perl et al. 2015). It is possible that not all mapping in this case can be considered vulnerable in relation to a CVE ID. However, from the evidence of our statistical tests, we believe our conclusions help support the internal validity.

Regarding usage of functions, we acknowledge that a vulnerability may span multiple functions within a single source code file or multiple files. Given the motivation of our work,

we are foregoing any contextual aspect of the function which may break such a semantic link (syntax) within a file or across files. In our exclusive scenario, we are focused on identifying any vulnerability that may be present. To that end our results may not hold in other cases, and we acknowledge that our validity may be limited to this scenario only.

Given the variety of code snippets that may be available on platforms such as Stack Overflow, these might not always be provided in the form of a function. We acknowledge that our approach of identifying, extracting and pairing source code based on functions may not be the best approach given our motivation. However, we pursued this method in order to provide a structure in which we could avoid the class imbalance problem by creating equal amounts of "fixed" and "vulnerable" data. Without a foundation, we assumed we would deal with a lot of noise and randomness throughout the datasets. Our goal was to simplify the data and focus solely on modifications which lead to or mitigated vulnerabilities. To that end, our approach may not be applicable in other situations nor might our results hold.

During our data collection and extraction process, we utilize a series of commits which correspond to "fixed" and "vulnerable" instances of source code files which have been modified. In doing so, we may backtrack from a provided "fixing" commit and retrieve data from the parent commit, which we acknowledge as still containing a vulnerability. For our study, we are not claiming the parent commit introduced the vulnerability, but assume it contains an instance in which a vulnerability is present. We do acknowledge that it is possible for developers to include irrelevant changes within a commit or create a series of commits during the process of mitigating a vulnerability. However, currently, we are not evaluating such events.

During our function extraction process, we apply a lightweight approach for identifying available functions in source code files by constructing a series of regular expressions and conditions. Given various developer coding styles, it is possible that the established regular expressions and conditions formed in the extraction process do not meet all possibilities for finding available functions within source code files. As a result, we cannot claim to find every single function across all source code files for all projects. Given the number of functions we were able to extract and pair (over 199,000 individual functions and over 99,000 pairings), we assume that any data which was discarded would not have significant changes to our results. More preprocessing and conditions could be added to the process to further expand its capabilities of extracting available functions which can potentially increase the function dataset. The regular expressions used could also be improved upon.

For our processing methodology, we forgo any contextual aspect of the source code regarding our motivation in detecting vulnerabilities as they may appear in code snippets. As such, we may discard any information we do not deem relevant but could be utilized in other scenarios. For example, during our data processing we choose to ignore symbols that appear within the source code. With our usage of natural language processing (NLP) techniques, features are extracted and used to train the models. With these techniques, symbols are not considered features. We acknowledge this may break any supposed syntax or semantic of source code, but usage of NLP techniques was necessary given our goal of using text-based approaches. To that end, anything the NLP techniques do not regard as a feature, by default are discarded.

In relation to discarding symbols, the process of generalizing them into textual representations (i.e., = means "equal") as well as variable names was considered. This approach was done by Li et al. (2018) in creating symbolic representation of their defined code gadgets from a semantic standpoint. Although we experimented with similar approaches in trials prior to our preliminary experiment, we did not elect to use such methods given our difference in scenarios. However, we anticipate that such additional processing would not change the outcome of our conclusions nor the answers to our research questions. We assume that

the insignificance of such occurrences, such as variable names only being used once, would not be enough to influence the training of a model.

It could be assumed that an unbalanced distribution of function pairs may affect the conclusion of our research questions due to the top three projects containing the most function pairings and features. However, our results in Tables 17 and 21 do not reflect such suspicions. If our models were heavily trained by the top three projects, then the prediction results when cross testing among them (i.e., P1-P2, P1-P3) would arguably be better than the results compared to projects with less pairings and features (i.e., P1-P7, P1-P8, etc). Given our methods to answer our research questions related to the training and testing within and across projects, we only apply trained models which are specific to a single project, thus the features from other projects would not be learned. Additionally, if we observe vulnerability prediction from a real world scenario, occurrences of some vulnerabilities may be more frequent compared to others. In that case, such an uneven distribution as presented here conforms to such a scenario.

8.3 External Validity

External validity comes from the ability to generalize the accuracy of findings (Bates and Cozby 2017). Source code is best understood through documentation or a given context of a software system it is integrated into. However, from our motivation of detecting vulnerabilities via non-executable code snippets, available on platforms such as Stack Overflow, we elect to forego such contextual aspect of source code. Due to the limited context throughout the extracted function pairs representing "fixed" and "vulnerable" instances of source code, it is possible that the functions utilized in the machine learning training and testing are not sufficient to determine an appropriate distinction of classification. Interpreting a piece of source code through text-based analysis with natural language processing techniques, might not provide the best possible result for a different software system or programming language. However, from our overall conclusion, we expect that the external validity of our research to hold and our research question answers to remain true. Additionally, the database dump we utilize may be considered outdated, but our goal was not to predict new vulnerabilities from this data. To combat this, we included an additional dataset of C/C++ vulnerability data spanning from 2002-2019.

8.4 Reliability

Reliability refers to the stability of a measurement (Bates and Cozby 2017). Given our own defined methods of data processing throughout the paper, it is possible for someone to replicate the work. However their methods might be different and cause inconsistent results related to the data used or metrics generated. For example, the machine learning models and natural language processing technique implementations have various optional parameters which can be set aside from the default values used. In our case, we elected to utilize the out-of-the-box implementations with default values (cf. Section 4.2).

Given the second dataset, Big-Vul (Fan et al. 2020), was smaller than the first dataset, VCCFinder (Perl et al. 2015), we elected to merge the two based on their similarities regarding vulnerabilities found within C/C++ projects. We do acknowledge that scores could be affected to a certain extent. However, based on the experiments involving the merged data, our research questions and answers did not change. Additionally, examining the two datasets based on other factors such as dividing by year may provide interesting results. To that end, we plan to explore this in future work to determine if new insight could be provided.

9 Conclusion and Future Work

This paper provides an empirical study on text-based machine learning models for vulnerability detection across applicable data from 344 projects with a total of 2,182 vulnerabilities and 38 vulnerability types. Our motivation and reason for using text-based machine learning models stem from the prevalence of vulnerable code snippets which may be present on crowdsourced Q&A platforms such as Stack Overflow. In such cases, snippets of code which lack context may be presented as an answer, but also inadvertently contain a vulnerability. With motivation to combat this problem, we establish a foundation in which machine learning models can be used. Given the variety of characteristics regarding vulnerabilities and systems, we evaluated two vulnerability datasets to determine instances of "fixed" and "vulnerable" source code files related to vulnerabilities as identified by CVE IDs. To mimic the presence of code snippets, we extracted functions from these files and created equal pairings to avoid the class imbalance problem.

We evaluated this data using seven machine learning models, five natural language processing techniques, and three defined data processing methods. We conducted a preliminary experiment coupled with statistical analysis to determine an appropriate combination towards vulnerability detection efforts. From the results of our preliminary experiment, we selected the Convolutional Neural Network (CNN) model, the Term Frequency-Inverse Document Frequency (TF-IDF) natural language processing (NLP) technique, and our own data processing method with reduced features named *No Context* (NC). From this combination, we performed further experiments which trained and tested a series of CNN models within and across the top 10 projects and top 10 CWE vulnerabilities types based on the amount of extracted function pairings we processed. From these experiments, we were able to answer our research questions and found that text-based vulnerability detectors are not effective in detecting vulnerabilities across projects and CWE vulnerability types.

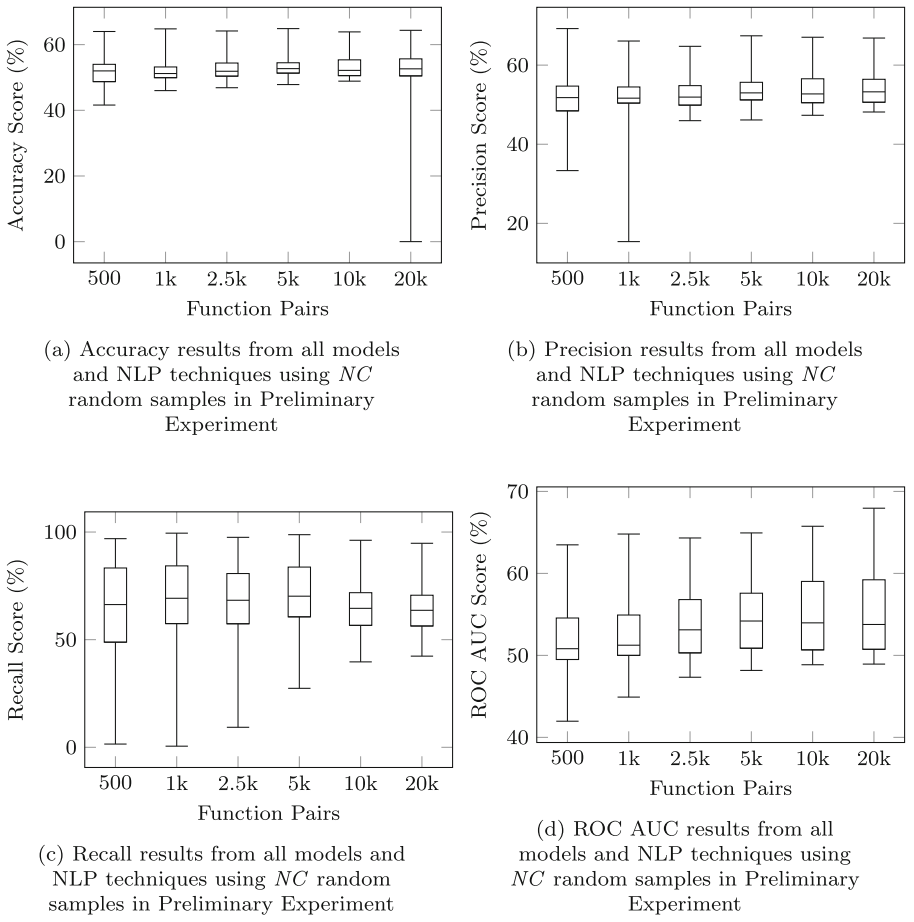
Our contributions include increasing the scope of measurement by performing empirical statistical analysis, determining if there is a statistically significant difference between within versus across prediction, and providing insight for possible reasoning behind the results for within versus across testing. Compared to related works, we utilize larger datasets composed of various systems and vulnerabilities, avoid issues related to the class imbalance problem by creating equal function pairings between fixed and vulnerable samples, and perform extensive training and testing within and across various projects and CWE vulnerability types.

We further describe potential reasons as to why such models may not be effective. For example, the overlap of features used to train text-based machine learning models across projects and vulnerability types are low. From this, we assume that specific features are unique to certain projects or vulnerability types. Additionally, certain features may carry more weight regarding a specific project or vulnerability type and may not be applicable when applying across. Thus, some features may not be applicable in cross-testing. Although the overlap of features across projects and vulnerability types might be low, additional analysis could be performed to determine a correlation between the "fixed" and "vulnerable" features that occur within a project. It is possible that with a high overlap, model confusion could occur and ultimately lead to inadequate results. We can assume that additional factors may also play a role. Further exploration is needed in determining an approach for identifying how similarities or differences between features can affect such models.

In this study, we focus on C/C++ related source code, but additional research could expand on analysis methods by utilizing other programming languages and projects. Another approach might be to consider using trained features from one type of programming

language and apply to another. Further exploration is needed into why this approach (specifically using text-based machine learning models on source code for vulnerability detection) does not work. Exploring different factors regarding the datasets used may provide alternative avenues which could yield interesting results, such as dividing by year. Future research is also encouraged to construct a new dataset with recent projects and vulnerabilities to determine if our approach is applicable since portions of used datasets may be considered outdated. Finally, research regarding the explainability of machine learning models as it relates to vulnerability prediction is encouraged. To fully understand potential solutions to such a problem, we must first be able to decipher the reasoning of models and how the data is truly being interpreted and used towards vulnerability prediction in source code. We plan to explore such ideas in future work.

Appendix A: Preliminary Experiment Additional Metrics



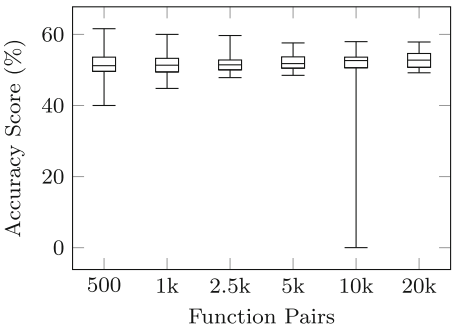
(a) Accuracy results from all models and NLP techniques using *NC* random samples in Preliminary Experiment

(b) Precision results from all models and NLP techniques using *NC* random samples in Preliminary Experiment

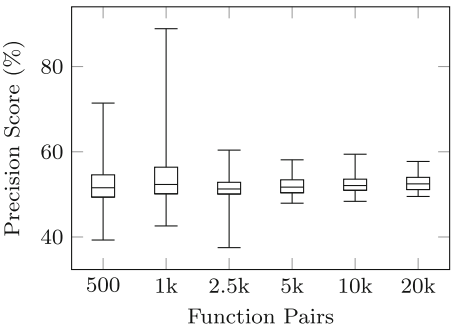
(c) Recall results from all models and NLP techniques using *NC* random samples in Preliminary Experiment

(d) ROC AUC results from all models and NLP techniques using *NC* random samples in Preliminary Experiment

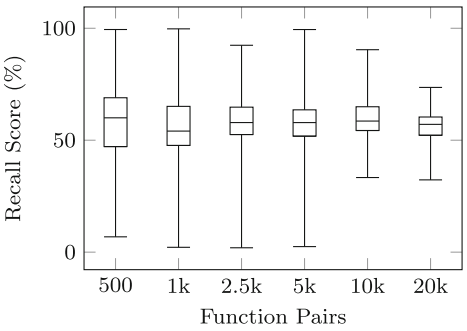
Fig. 6 Additional metrics from all models and NLP techniques using *NC* random samples in Preliminary Experiment



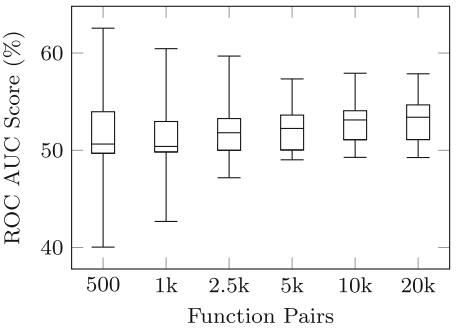
(a) Accuracy results from all models and NLP techniques using LC random samples in Preliminary Experiment



(b) Precision results from all models and NLP techniques using LC random samples in Preliminary Experiment



(c) Recall results from all models and NLP techniques using LC random samples in Preliminary Experiment



(d) ROC AUC results from all models and NLP techniques using LC random samples in Preliminary Experiment

Fig. 7 Additional metrics from all models and NLP techniques using *LC* random samples in Preliminary Experiment

Data Availability The datasets generated during and/or analyzed during the current study are available in the "emse_data" repository, https://github.com/krn65/emse_data

Declarations

Conflict of Interests The authors of this manuscript have no conflicts of interest.

References

Abdalkareem R, Shihab E, Rilling J (2017) On code reuse from stackoverflow: An exploratory study on android Apps. Inf Softw Technol 88:148–158. https://doi.org/10.1016/j.infsof.2017.04.005

Ban X, Liu S, Chen C, Chua C (2019) A performance evaluation of deep-learnt features for software vulnerability detection. Concurr Comput Pract Experience 31(19):e5103. https://doi.org/10.1002/cpe.5103

Bates S, Cozby P (2017) Methods in behavioral research. McGraw-Hill Education, New York

Cavusoglu H, Mishra B, Raghunathan S (2004) The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers. Int J Electron Commer 9(1):70–104. https://doi.org/10.1080/10864415.2004.11044320

Chen Y (2015) Convolutional neural network for sentence classification. Master's thesis, University of Waterloo. http://hdl.handle.net/10012/9592

Chernis B, Verma R (2018) Machine learning methods for software vulnerability detection. In: Proceedings of the 4th ACM international workshop on security and privacy analytics, pp 31–39. https://doi.org/10.1145/3180445.3180453

Cor K, Sood G (2018) Pwned: How often are Americans' online accounts breached? arXiv:1808.01883

Czerwonka J, Greiler M, Tilford J (2015) Code reviews do not find bugs. How the current code review best practice slows us down. In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 2. IEEE, pp 27-28. https://doi.org/10.1109/ICSE.2015.131

Devlin J, Chang MW, Lee K, Toutanova K (2018) Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:181004805, https://doi.org/10.48550/arXiv.1810.04805

Dowd M, McDonald J, Schuh J (2006) The art of software security assessment: Identifying and preventing software vulnerabilities. Pearson Education

Duan X, Wu J, Ji S, Rui Z, Luo T, Yang M, Wu Y (2019) Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In: IJCAI, pp 4665–4671. https://doi.org/10.24963/ijcai.2019/648

Dunn OJ (1961) Multiple comparisons among means. J Am Stat Assoc 56(293):52–64. https://doi.org/10.1080/01621459.1961.10482090

Egele M, Scholte T, Kirda E, Kruegel C (2008) A survey on automated dynamic malware-analysis techniques and tools. ACM Comput Surv (CSUR) 44(2):1–42. https://doi.org/10.1145/2089125.2089126

Fan J, Li Y, Wang S, Nguyen TN (2020) AC/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th international conference on mining software repositories, pp 508–512. https://doi.org/10.1145/3379597.3387501

Fischer F, Böttinger K, Xiao H, Stransky C, Acar Y, Backes M, Fahl S (2017) Stack overflow considered harmful? the impact of copy&paste on android application security. In: 2017 IEEE symposium on security and privacy (SP). IEEE, pp 121–136. https://doi.org/10.1109/SP.2017.31

Ghaffarian SM, Shahriari HR (2017) Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput Surv (CSUR) 50(4):1–36. https://doi.org/10.1145/3092566

Grieco G, Grinblat GL, Uzal L, Rawat S, Feist J, Mounier L (2016) Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the 6th ACM conference on data and application security and privacy, pp 85–96. https://doi.org/10.1145/2857705.2857720

Harer JA, Kim LY, Russell RL, Ozdemir O, Kosta LR, Rangamani A, Hamilton LH, Centeno GI, Key JR, Ellingwood PM et al (2018) Automated software vulnerability detection with machine learning. arXiv:1803.04497

Hovsepyan A, Scandariato R, Joosen W, Walden J (2012) Software vulnerability prediction using text analysis techniques. In: Proceedings of the 4th international workshop on Security measurements and metrics, pp 7–10. https://doi.org/10.1145/2372225.2372230

Huang S, Tang H, Zhang M, Tian J (2010) Text clustering on national vulnerability database. In: 2010 2nd international conference on computer engineering and applications, vol 2. IEEE, pp 295–299. https://doi.org/10.1109/ICCEA.2010.209

Ijaz M, Durad MH, Ismail M (2019) Static and dynamic malware analysis using machine learning. In: 2019 16th international BHURBAN conference on applied sciences and technology (IBCAST). IEEE, pp 687–691. https://doi.org/10.1109/IBCAST.2019.8667136

Jie G, Xiao-Hui K, Qiang L (2016) Survey on software vulnerability analysis method based on machine learning. In: 2016 IEEE 1st international conference on data science in cyberspace (DSC). IEEE, pp 642–647. https://doi.org/10.1109/DSC.2016.33

Kim J, Hubczenko D, Montague P (2019) Towards attention based vulnerability discovery using source code representation. In: International conference on artificial neural networks. Springer, pp 731–746. https://doi.org/10.1007/978-3-030-30490-4_58

Kim Y (2014) Convolutional neural networks for sentence classification. In: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). Association for Computational Linguistics, Doha, Qatar, pp 1746–1751. https://doi.org/10.3115/v1/D14-1181, https://aclanthology.org/D14-1181

Klock R (2021) Quality of SQL code security on stackoverflow and methods of prevention. PhD thesis, Oberlin College. http://rave.ohiolink.edu/etdc/view?acc_num=oberlin1625831198110328

Koroteev M (2021) Bert: A review of applications in natural language processing and understanding. arXiv:2103.11943

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621. https://doi.org/10.1080/01621459.1952.10483441
Layton R, Watters PA (2014) A methodology for estimating the tangible cost of data breaches. J Inf Secur Appl 19(6):321–330. https://doi.org/10.1016/j.jisa.2014.10.012
Le QV, Mikolov T (2014) Distributed representations of sentences and documents. https://doi.org/10.48550/arXiv.1405.4053
Li P, Cui B (2010) A comparative study on software vulnerability static analysis techniques and tools. In: 2010 IEEE international conference on information theory and information security. IEEE, pp 521–524. https://doi.org/10.1109/ICITIS.2010.5689543
Li X, Chang X, Board JA, Trivedi KS (2017) A novel approach for software vulnerability classification. In: 2017 annual reliability and maintainability symposium (RAMS). IEEE, pp 1–7. https://doi.org/10.1109/RAM.2017.7889792
Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y (2018) Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv:180101681, https://doi.org/10.14722/ndss.2018.23158
Li Z, Zou D, Xu S, Chen Z, Zhu Y, Jin H (2021a) Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans Dependable Sec Comput. https://doi.org/10.1109/TDSC.2021.3076142
Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z (2021b) Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans Dependable Secur Comput. https://doi.org/10.1109/TDSC.2021.3051525
Lin G, Zhang J, Luo W, Pan L, Xiang Y (2017) Poster: Vulnerability discovery with function representation learning from unlabeled projects. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2539–2541. https://doi.org/10.1145/3133956.3138840
Lin G, Zhang J, Luo W, Pan L, De Vel O, Montague P, Xiang Y (2019) Software vulnerability discovery via learning multi-domain knowledge bases. IEEE Trans Dependable Sec Comput. https://doi.org/10.1109/TDSC.2019.2954088
Lin G, Wen S, Han QL, Zhang J, Xiang Y (2020) Software vulnerability detection using deep neural networks: A survey. Proc IEEE 108(10):1825–1848. https://doi.org/10.1109/JPROC.2020.2993293
Liu B, Shi L, Cai Z, Li M (2012) Software vulnerability discovery techniques: A survey. In: 2012 4th international conference on multimedia information networking and security. IEEE, pp 152–156. https://doi.org/10.1109/MINES.2012.202
Liu S, Lin G, Han QL, Wen S, Zhang J, Xiang Y (2019) Deepbalance: Deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans Fuzzy Syst 28(7):1329–1343. https://doi.org/10.1109/TFUZZ.2019.2958558
Liu S, Lin G, Qu L, Zhang J, De Vel O, Montague P, Xiang Y (2020) CD-VulD: Cross-domain vulnerability discovery based on deep domain adaptation. IEEE Trans Dependable Secur Comput. https://doi.org/10.1109/TDSC.2020.2984505
Mäntylä V, Lassenius C (2008) What types of defects are really discovered in code reviews? IEEE Trans Softw Eng 35(3):430–448. https://doi.org/10.1109/TSE.2008.71
McQueen MA, McQueen TA, Boyer WF, Chaffin MR (2009) Empirical estimates and observations of 0day vulnerabilities. In: 2009 42nd Hawaii international conference on system sciences. IEEE, pp 1–12. https://doi.org/10.1109/HICSS.2009.186
Mikolov T, Chen K, Corrado G, Dean J (2013a) Efficient estimation of word representations in vector space. https://doi.org/10.48550/arXiv.1301.3781
Mikolov T, Sutskever I, Chen K, Corrado G, Dean J (2013b) Distributed representations of words and phrases and their compositionality. https://doi.org/10.48550/arXiv.1310.4546
Mokbal FMM, Dan W, Imran A, Jiuchuan L, Akhtar F, Xiaoxi W (2019) MLPXSS: an integrated XSS-based attack detection scheme in web applications using multilayer perceptron technique. IEEE Access 7:100567–100580. https://doi.org/10.1109/ACCESS.2019.2927417
Mubarek AM, Adalı E (2017) Multilayer perceptron neural network technique for fraud detection. In: 2017 international conference on computer science and engineering (UBMK). IEEE, pp 383–387. https://doi.org/10.1109/UBMK.2017.8093417
Perl H, Dechand S, Smith M, Arp D, Yamaguchi F, Rieck K, Fahl S, Acar Y (2015) VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp 426–437. https://doi.org/10.1145/2810103.2813604
Pham NH, Nguyen TT, Nguyen HA, Nguyen TN (2010) Detection of recurring software vulnerabilities. In: Proceedings of the IEEE/ACM international conference on automated software engineering, pp 447–456. https://doi.org/10.1145/1858996.1859089
Piessens F (2002) A taxonomy of causes of software vulnerabilities in internet software. In: Supplementary Proceedings of the 13th international symposium on software reliability engineering. Citeseer, pp 47–52

Plachkinova M, Maurer C (2018) Security breach at target. J Inf Syst Educ 29(1):11–20. https://aisel.aisnet.org/jise/vol29/iss1/7

Qiu X, Sun T, Xu Y, Shao Y, Dai N, Huang X (2020) Pre-trained models for natural language processing: A survey. Sci China Technol Sci 63(10):1872–1897. https://doi.org/10.1007/s11431-020-1647-3

Scandariato R, Walden J, Hovsepyan A, Joosen W (2014) Predicting vulnerable software components via text mining. IEEE Trans Softw Eng 40(10):993–1006. https://doi.org/10.1109/TSE.2014.2340398

Shar LK, Briand LC, Tan HBK (2014) Web application vulnerability prediction using hybrid program analysis and machine learning. IEEE Trans Dependable Secur Comput 12(6):688–707. https://doi.org/10.1109/TDSC.2014.2373377

Shin Y, Williams L (2008) An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the 2nd ACM-IEEE international symposium on Empirical software engineering and measurement, pp 315–317. https://doi.org/10.1145/1414004.1414065

Shu X, Tian K, Ciambrone A, Yao D (2017) Breaking the target: An analysis of target data breach and lessons learned. arXiv:1701.04940

Spanos G, Angelis L, Toloudis D (2017) Assessment of vulnerability severity using text mining. In: Proceedings of the 21st Pan-Hellenic conference on informatics, pp 1–6. https://doi.org/10.1145/3139367.3139390

Spreitzenbarth M, Schreck T, Echtler F, Arp D, Hoffmann J (2015) Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. Int J Inf Secur 14(2):141–153. https://doi.org/10.1007/s10207-014-0250-0

Su W, Yuan Y, Zhu M (2015) A relationship between the average precision and the area under the ROC curve. In: Proceedings of the 2015 international conference on the theory of information retrieval, pp 349–352. https://doi.org/10.1145/2808194.2809481

Sultana KZ, Deo A, Williams BJ (2016) A preliminary study examining relationships between nano-patterns and software security vulnerabilities. In: 2016 IEEE 40th annual computer software and applications conference (COMPSAC), vol 1. IEEE, pp 257–262. https://doi.org/10.1109/COMPSAC.2016.34

Tang G, Meng L, Wang H, Ren S, Wang Q, Yang L, Cao W (2020) A comparative study of neural network techniques for automatic software vulnerability detection. In: 2020 international symposium on theoretical aspects of software engineering (TASE). IEEE, pp 1–8. https://doi.org/10.1109/TASE49443.2020.00010

Telang R, Wattal S (2007) An empirical analysis of the impact of software vulnerability announcements on firm stock price. IEEE Trans Softw Eng 33(8):544–557. https://doi.org/10.1109/TSE.2007.70712

Wang H, Ye G, Tang Z, Tan SH, Huang S, Fang D, Feng Y, Bian L, Wang Z (2020) Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans Inf Forensics Secur 16:1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

Wang P, Johnson C (2018) Cybersecurity incident handling: A case study of the equifax data breach. Issues Inf Syst 19(3). https://doi.org/10.48009/3_iis_2018_150-159

Wijayasekara D, Manic M, McQueen M (2014) Vulnerability identification and classification via text mining bug databases. In: IECON 2014-40th annual conference of the IEEE industrial electronics society. IEEE, pp 3612–3618. https://doi.org/10.1109/IECON.2014.7049035

Yamaguchi F, Lindner F, Rieck K (2011) Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In: Proceedings of the 5th USENIX conference on Offensive technologies, pp 13–13. https://dl.acm.org/doi/10.5555/2028052.2028065

Zhang H, Wang S, Li H, Chen THP, Hassan AE (2021) A study of C/C++ code weaknesses on stack overflow. IEEE Trans Softw Eng. https://doi.org/10.1109/TSE.2021.3058985

Zhu M (2004) Recall, precision and average precision. Department of Statistics and Actuarial Science, University of Waterloo. Waterloo 2(30):6

Zhu Y, Kiros R, Zemel R, Salakhutdinov R, Urtasun R, Torralba A, Fidler S (2015) Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In: Proceedings of the IEEE international conference on computer vision, pp 19–27. https://doi.org/10.1109/ICCV.2015.11

Zou D, Wang S, Xu S, Li Z, Jin H (2019) μvuldeepecker: A deep learning-based system for multiclass vulnerability detection. IEEE Trans Dependable Secur Comput. https://doi.org/10.1109/TDSC.2019.2942930